



*High-performance Computing and Simulation (HCS)
Research Laboratory*



Grant Number N00014-97-1-0229
to the University of Florida
January - December 1997

***“Parallel and Distributed Computing Architectures and
Algorithms for Fault-Tolerant Sonar Arrays”***

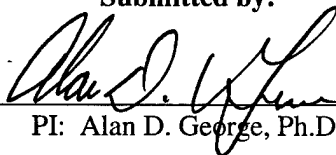
Annual Report #2

Submitted to the:
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5660

February 3, 1998

Attention:
Dr. Donald Davison
Dr. Charles Gaumond
ONR 321

Submitted by:


PI: Alan D. George, Ph.D.

Assoc. Prof. of ECE and Dir., HCS Research Lab
Dept. of Electrical and Computer Engineering
University of Florida
PO Box 116200, 327 Larsen Hall
Gainesville, FL 32611-6200
Phone: (352)392-5225, e-mail: george@hcs.ufl.edu

Contributing Research Team Members:
Alan George, Ryan Fogarty, Jesus Garcia, Ken Kim,
Jeff Markwell, Michael Miars, and Shonda Walker

in collaboration with Dr. Warren Rosen at Drexel University

19980217 542

DISTRIBUTION STATEMENT A

**Approved for public release
Distribution Unlimited**

Table of Contents

Table of Contents.....	ii
List of Figures	v
List of Tables.....	viii
List of Acronyms.....	ix
1. Introduction	1
1.1. Technical Background	1
1.2. Technical Issues.....	1
1.2.1. Topology, Architecture, and Protocol.....	2
1.2.2. Algorithm Decomposition, Partitioning, and Mapping.....	2
1.2.3. Fault Tolerance	3
1.2.4. Simulator Tools	3
1.3. Technical Approach.....	4
1.4. Summary of Results.....	4
1.5. Tasks.....	5
Task 1. Topology, Architecture, and Protocol Development	5
Task 2. Algorithm Development and Modeling	6
Task 3. Simulator Development	6
Task 4. Preliminary Software System Development	6
Task 5. Strawman Node Circuit Design	6
Task 6. Algorithm and Preliminary Software Development for Split-Aperture Beamforming with Cross-Spectral Correlation.....	7
Task 7. Hardware Prototype Development.....	7
Task 8. Detailed Software System Development	7
2. Integrated Simulation Environment	8
2.1. Foundation Components	8
2.2. Structure of Environment.....	9
2.2.1. Parallel Process Code	10
2.2.2. MPI runtime system.....	12
2.2.3. ISE Version 1.3 Extensions	13
2.2.3.1. Multiple Iterations.....	13
2.2.3.2. Runtime System.....	14
2.2.3.3. Trigger Network	15
2.3. 'C54 Simulator	15
2.4. High-fidelity Network Models.....	15
2.4.1. Slotted Ring	16
2.4.2. Test Applications	18
2.4.3. Ratchet Network	22
2.5. DPSA Architecture and Algorithm Performance Predictions.....	24
2.6. Conclusions	27
3. Parallel Conventional Beamforming.....	28
3.1. Sequential Conventional Beamforming.....	28
3.2. Parallel Conventional Beamforming Algorithms.....	29
3.2.1. Coarse-grained Parallel Algorithms.....	30
3.2.1.1. Coarse-grained Unidirectional Algorithms	31
3.2.1.2. Coarse-grained FCBDN Algorithms.....	32
3.2.2. Medium-grained Parallel Algorithms	34
3.2.2.1. Medium-grained Unidirectional Algorithm	34
3.2.2.2. Medium-grained FCBDN Algorithm.....	35
3.3. Performance Prediction	39

3.4. Performance Results	46
3.5. Conclusions	50
4. Advanced Beamforming Algorithms.....	51
4.1. The Split-Aperture Conventional Beamforming Algorithm	51
4.2. Computer Simulation.....	53
4.2.1. Intermediate Data Products.....	54
4.3. Performance Analysis.....	56
4.3.1. SA-CBF vs. CBF	56
4.3.2. Parallel Decompositions of SA-CBF	57
4.4. ABF Description.....	58
4.5. Conclusions	62
5. Prototype Hardware Architecture	63
5.1. Prototype Hardware Considerations	63
5.2. Prototype Processor and Basic Node Architecture	64
5.2.1. Overview of the Node Processor	64
5.2.2. Functional Overview of the Development Board	66
5.3. Prototype System Configuration.....	67
5.4. Testing and Simulation Environment	68
5.4.1. In-circuit Emulator and Debugging Tool.....	70
5.5. Conclusions	71
Conclusions	72
Bibliography.....	74
Appendix A. Extensions to Integrated Simulation Environment.....	81
A.1. Integrated Simulation Environment Internals	81
A.1.1. Communication Structures.....	81
A.1.2. Process Structure and Spawning	86
A.1.3. BONEs Interface Structure and Spawning.....	88
A.1.4. Communication and Relays	89
A.1.5. Portal Internals.....	92
A.1.5.1. Decomposition of ISE Functions	92
A.1.5.2. Enhancing ISE Performance	95
A.2. User's Guide	97
A.2.1. Minimum Hardware Requirements.....	97
A.2.2. Minimum Software Requirements	97
A.2.3. Starting the ISE Program	97
A.2.4. Starting the Network Simulation.....	98
A.2.5. After the Simulation Has Started	99
A.2.6. Tips for Advanced Users	99
A.2.7. Requirements of the MPI program.....	99
A.3. Implementation Guide	99
A.3.1. Portal.....	99
A.3.2. Hooking up the Portal	102
A.4. Currently Supported MPI Functions in ISE	104
A.5. Stages of ISE Development	105
A.6. Other SCALE BONEs/MPI Notes	105
A.7. Sample Remote Host File for ISE.....	106
Appendix B. Extensions to Parallel Conventional Beamforming.....	107
B.1. Extended Work in Time-Domain Beamforming	107
B.1.1. Non-Linear Algorithmic Enhancements for Time-Domain Beamformers.....	107
B.1.1.1. Circular Shift Method	108

B.1.1.2. Cycle Check Procedure for Circular Shifting.....	111
B.1.1.3. Fractional Beamsteering.....	113
B.1.1.4. Bresenham's Algorithm for Fractional Beamsteering.....	115
B.1.2. The Butterfly Beamformer.....	116
B.1.2.1. The Butterfly Characteristic.....	116
B.1.2.2. Complexity of the Butterfly Beamformer.....	118
B.1.2.3. The Delicate Butterfly.....	119
B.1.2.4. A Global Data Scope Time-Domain Beamform Algorithm (GDS-TDBA).....	119
B.1.2.5. The GDS-TDBA Algorithm.....	120
B.1.2.6. GDS-TDBA Memory Matrix Solution.....	121
B.1.2.7. GDS-TDBA Parallelism.....	122
B.1.2.8. GDS-TDBA Source Code.....	124
B.2. Extended Work In FFT Conventional Beamforming.....	124
B.2.1. Pipelined Medium-Grain Full-Capability Algorithm.....	125
B.2.2. Category Definitions.....	129
B.2.3. Category Performance Results.....	131
Appendix C. Extensions to Advanced Beamforming Algorithms.....	133
C.1. Details of Algorithm Stages.....	133
C.1.1. Fast Fourier Transform Stage.....	133
C.1.2. Frequency-Domain Beamforming Stage.....	133
C.1.3. Cross-correlation Stage.....	135
C.1.4. Mapping Process Stage.....	137
C.2. Computer Simulation.....	139
C.2.1. Parameters Used in the Simulation.....	139
C.2.2. Generate Input Data.....	139
C.2.3. Complex Situations.....	140
Appendix D. Extensions to Prototype Hardware Architecture.....	144
D.1. Additional Node Processor Features.....	144
D.2. TDM Serial Port.....	144
D.2.1. TDM Serial Port Registers.....	145
D.2.2. TDM Serial Port Operation.....	146
D.2.3. Receive/Transmit Operations.....	147
Appendix E. Fault-tolerant Architectures and Algorithms.....	149
E.1. Fault-Tolerant Services.....	149
E.2. Data Padding Samples.....	152
E.3. Automatic Repeat Requests.....	154
E.4. Reinitialize Request.....	154
E.5. Ping (Check Status) Request.....	155
E.6. Beamform Request.....	156
E.7. Arbitrate New Master.....	157
E.8. Verification of GDS Fault-Tolerant Kernel.....	158
E.9. Conclusions.....	162

List of Figures

Figure 2.1 – ISE Runtime Environment	9
Figure 2.2 – Global Data Scope (GDS) versus Local Data Scope (LDS) Systems	16
Figure 2.3 - Input Switch.	17
Figure 2.4 - Slot Usage.	18
Figure 2.5 - Fill Empty Slot.	18
Figure 2.6 - Throughput versus Slot Sizes.	20
Figure 2.7 - Throughput of 8 Node Cluster versus Number of Slots.	21
Figure 2.8 – 100-Barriers Test.	21
Figure 2.9 – Ratchet Network Protocol	22
Figure 2.10 – Execution Time for Non-pipelined Program over Different Network and Processor Speeds	25
Figure 2.11 – Execution Time for Pipelined Program over Different Network and Processor Speeds	25
Figure 3.1 - Flowchart for Sequential FFT Beamformer.	29
Figure 3.2 - Flowchart for CPUNF, Coarse-grained Unidirectional Network Independ. FFT Beamformer.	31
Figure 3.3 - Flowchart for the CPNF, Coarse-grained Network Independent FFT Beamformer.	33
Figure 3.4 - Flowchart for the MPUF, Medium-grain Unidirectional FFT Beamformer	35
Figure 3.5 - Flowchart for MPRF, Medium-grain Full-capability FFT Beamformer with Rudimentary Ring Simulation.	36
Figure 3.6 - Ring Communication for MPRF, Medium-grain Full-capability FFT Beamformer.	37
Figure 3.7 - Flowchart for MPBF, Medium-grain Full-capability FFT Beamformer with Rudimentary Bidirectional Array Simulation.	38
Figure 3.8 - Bidirectional Array Communication for MPBF, Medium-grain Full-capability FFT Beamformer.	39
Figure 3.9 – Floating-point Operations per Iteration for Parallel Beamformers.	42
Figure 3.10 – Floating-Point Communications per Iteration for Parallel Beamformers.	44
Figure 3.11 - Expected Execution Times.	45
Figure 3.12 - Predicted Execution Times versus Number of Steering Directions and Number of Nodes	46
Figure 3.13 - Speedups versus Number of Nodes.	47
Figure 3.14 - Efficiencies of Programs versus Number of Nodes.	48
Figure 3.15 - Speedups versus Number of Steering Directions	49
Figure 3.16 - Efficiencies versus Number of Steering Directions	49
Figure 4.1 - Block Diagram for the Split-Aperture Conventional Beamforming Algorithm.	52
Figure 4.2 – Flowchart of the SA-CBF.	54
Figure 4.3 - Steered to the data incoming angle (47.8°); (a) subarray1 beamforming output; (b) subarray2 beamforming output; and (c) cross-correlation without SCOT and normalization.	55
Figure 4.4 - Steered to the -90° (data angle is 47.8°): (a) subarray1 beamforming output; (b) subarray2 beamforming output; and (c) cross-correlation without SCOT and normalization.	55
Figure 4.5 - Effect of SCOT and Normalization: (a) Figure 4.4c redone with normalization and (b) Figure 4.4c redone with SCOT and normalization.	56
Figure 4.6 – SA-CBF output with 15 sub-array steering angles and 57 output angles: (a) with linear interpolation and (b) with τ -interpolation.	56
Figure 4.7 – Decompositions of the SA-CBF Algorithm.	58
Figure 4.8 – Beam pattern of the SA-CBF with a uniform shading (no window applied)	59
Figure 4.9 – Diagram for the ABF weight with frequency averaging.	60
Figure 4.10 – Flowchart for SA-ABF.	61
Figure 5.1 - A functional diagram of the TI TMS320C54x Architecture illustrating multiple functional units [TMS97A].	65
Figure 5.2 – A Board Diagram of DSKplus detailing the components on the board [TMS96].	67
Figure 5.3 – Prototype System Configuration	68
Figure 5.4 - DSP Software Development Steps [TMS97C]	70
Figure A.1 – Structure of the ISE Bridge File	82
Figure A.2 – Structure of the ISE Share File.	83
Figure A.3 – Structure of the ISE BONEs File.	85
Figure A.4 - The BONEs_MPI Portal	93

Figure A.5 - Create DPSA Data Packet Function. This function creates the data structure for the BOnES_MPI Data Packet.....	94
Figure A.6 - Close-up of Receiver Matching Logic.	95
Figure A.7 - Close-up of the Six-Node Bidirectional Array's Global Clock Gate	96
Figure A.8 - Master Clock with the ClockRouter Gate.	97
Figure A.9 - BOnES/MPI Portal over Two Network Protocol Stacks	100
Figure A.10 - Portal Input Queue.....	101
Figure A.11 - System View for an ISE-Capable Network	102
Figure A.12 - Internals of the Two Versions of the Portal.....	103
Figure A.13 - Portal Data Structures	104
Figure A.14 - Sample .rhost File.	106
Figure B.1 - Beamformer Response versus Azimuth.....	109
Figure B.2 - Typical Incoming Signals	109
Figure B.3 - Incoming Signal and "Check_Cycle" Output	110
Figure B.4 - Circular Shift Method Results	111
Figure B.5 - Experiments with the "check_cycle" routine.....	113
Figure B.6 - Fractional Steering.....	114
Figure B.7 - Fraction Steering and Fractional Steering with Circular Shift Results	115
Figure B.8 - Interpolated Line of Bresenham's Algorithm [JOY97]	116
Figure B.9 - Butterfly Detailed Flowchart	117
Figure B.10 - Butterfly High-Level Flowchart	118
Figure B.11 - Requestor/Server Paradigm	120
Figure B.12 - Beamform Reduction Process	123
Figure B.13 - Beam Steering and the Reduction Operation.....	123
Figure B.14 - Original Medium-grain Full-capability Network-independ. FFT Beamformer Flowchart...	126
Figure B.15 - Pipelined Medium-grain Full-capability Network-independ. FFT Beamformer Flowchart.	126
Figure B.16 - Execution Times for Pipelined Medium-grain, Non-pipelined Medium-grain, and Coarse-grain Programs with 91 Steering Directions	127
Figure B.17 - Execution Times for Pipelined Medium-grain, Non-pipelined Medium-grain, and Coarse-grain Programs with 181 Steering Directions.....	127
Figure B.18 - Speedups for Pipelined Medium-grain, Non-pipelined Medium-grain, and Coarse-grain Programs with 91 Steering Directions	128
Figure B.19 - Speedups for Pipelined Medium-grain, Non-pipelined Medium-grain, and Coarse-grain Programs with 181 Steering Directions	128
Figure B.20 - Sequential Flowchart for the Frequency-domain Delay-and-Sum Beamformer.....	130
Figure B.21 - Execution Times for Sequential and Parallel Programs with 8 Nodes and 91 Steering Directions.....	131
Figure B.22 - Execution Times for Sequential and Parallel Programs with 8 Nodes and 181 Steering Directions.....	131
Figure C.1 - Geometry of the sub-array elements.....	134
Figure C.2 - Geometry of the whole array with 10 nodes.....	135
Figure C.3 - Illustration of the Cross-correlation Correctness.	136
Figure C.4 - Weight Functions for Composite Beam Correlation.....	137
Figure C.5 - Representation of Beamformer Angles, Steered Angles, and Interpolated Angles	137
Figure C.6 - Demonstration of the frequency and time domain relation.	140
Figure C.7 - Generated Input Data without Noise from 47.8 degrees.....	140
Figure C.8 - SA-CBF Output for Multiple Sources.	141
Figure C.9 - Input Signals with Noise and SA-CBF Outputs.....	142
Figure D.1 - TDM Serial Port Register Bits.	146
Figure D.2 - TDM Serial Port Wiring Diagram	146
Figure D.3 - TDM Register Contents.....	147
Figure D.4 - TDM Communication Scenario	148
Figure E.1 - System Diagram and Fault-Tolerant Services.	150
Figure E.2 - The Beamform Reduction Process.	151
Figure E.3 - Data Padding.....	153
Figure E.4 - Padding Data Streams.....	154

Figure E.5 - Reinitialize Request.....	155
Figure E.6 - The Ping Request.....	156
Figure E.7 - The Beamform Request.....	157
Figure E.8 - Master Arbitration.....	158
Figure E.9 - Efficiency Test.....	162

List of Tables

Table 2.1 – Proposed Packet Format	23
Table 3.1 – Summary of Algorithm Acronyms and Properties	30
Table 3.2 – Computational Building Blocks Used in Prediction Models	40
Table 3.3 - Parallel Decomposition Equations	41
Table 3.4 - Communication Functions	43
Table 3.5 - Communication Cost Equations	43
Table D.1 - TDM Serial Port Registers.	145

List of Acronyms

Acronym	Meaning
ABF	A daptive B eamforming
ACC	A uto- and C ross- C orrelation Beamforming
ADC	A nalog-to- D igital C onverter
AGC	A utomatic G ain C ontrol
ASCII	A merican S tandard C ode for Information Interchange
ASIC	A pplication- S pecific I ntegrated C ircuit
ATM	A synchronous T ransfer M ode
BDE	B lock D iagram E ditor (in BONeS)
BDN	B ounded- D egree N etwork
BDS	B asic D elay-and- S um Beamforming
BF	B eamforming
BONeS	B lock O riented N etwork S imulator
CBF	C onventional B eamforming
CC-NUMA	C ache- C oherent N on- U niform M emory A ccess
COMA	C ache- O nly M emory A ccess
COTS	C ommercial- O ff- T he- S helf
CSMA/CD	C arrier S ense M ultiple A ccess with C ollision D etection
D&C	D ivide A nd C onquer
DCE	D ata C ommunication E quipment
DFT	D iscrete F ourier T ransform
DLC (DLL)	D ata L ink C ontrol (D ata L ink L ayer)
DOA	D irection O f A rrival
DPSA	D istributed P arallel S onar A rray
DQDB	D istributed Q ueue, D ual B us
DRAM	D ynamic R AM
DSE	D ata S tructure E ditor (in BONeS)
DSI	D elay-and- S um with I nterpolation
DT2	D ecimation in T ime by 2
DTE	D ata T erminating E quipment
EDU	E lectrical D istribution U nit
FDM	F requency- D ivision M ultiplexing
FFT	F ast F ourier T ransform
FIFO	F irst I n / F irst O ut
FPGA	F ield P rogrammable G ate A rray
FTP	F ile T ransfer P rotocol
HCS	H igh-performance C omputing and S imulation R esearch L aboratory
HDLC	H igh- L evel D ata L ink C ontrol
GUI	G raphical U ser I nterface
GDS	G lobal D ata S cope
ICN	I nter C onnection N etwork
IEEE	I nstitute of E lectrical and E lectronic E ngineers
I/O	I nput / O utput
IP	I nternet P rotocol
ISO	I nternational S tandards O rganization

LAN	Local Area Network
LCC	Logical Link Control
LPF	Low-Pass Filter
LSB	Least Significant Bit
MAC	Medium Access Control
MAN	Metropolitan Area Network
MMIC	Microwave Monolithic Integrated Circuit
MPI	Message-Passing Interface
MSB	Most Significant Bit
MTTF	Mean Time To Failure
MUTEX (mutex)	MUTual EXclusion
NOW	Network Of Workstations
NUMA	Non-Uniform Memory Access
O/S	Operating System
OSI	Open Systems Interconnect
PBF	Parallel Bidirectional Frequency-Domain Beamforming
PBT	Parallel Bidirectional Time-Domain Beamforming
PC	Phase Center
PCMCIA	Personal Computer Memory Card International Association
PDC	Parallel and Distributed Computing
PLD	Programmable Logic Device
PNF	Parallel Network-independent Frequency-Domain Beamforming
PNT	Parallel Network-independent Time-Domain Beamforming
PRAM	Parallel Random Access Machine
PRF	Parallel Ring Frequency-Domain Beamforming
PRT	Parallel Ring Time-Domain Beamforming
PUF	Parallel Unidirectional Frequency-Domain Beamforming
PUT	Parallel Unidirectional Time-Domain Beamforming
PVM	Parallel Virtual Machine
RAM	Random Access Memory
RDSA	Rapidly Deployable Sonar Array
RISC	Reduced Instruction Set Architecture
SA-CBF	Split Aperture Convention Beamforming
SBF	Sequential Bidirectional Frequency-Domain Beamforming
SBT	Sequential Bidirectional Time-Domain Beamforming
SCI	Scalable Coherent Interface
SEQFFT	Purely SEQuential FFT Beamforming
SM	Simulation Manager (in BONeS)
SMP	Symmetric MultiProcessing
SRF	Sequential Ring Frequency-Domain Beamforming
SRT	Sequential Ring Time-Domain Beamforming
SNT	Sequential Network-independent Time-Domain Beamforming
SNF	Sequential Network-independent Frequency-Domain Beamforming
SPGA	System Programmable Gate Array
SRAM	Static RAM
SUF	Sequential Unidirectional Frequency-Domain Beamforming
SUT	Sequential Unidirectional Time-Domain Beamforming
TCP	Transmission Control Protocol

TDM	T ime- D ivision M ultiplexing
UMA	U niform M emory A ccess
VCSEL	V ertical C avity S urface E mitting L aser
WAN	W ide A rea N etwork

1. Introduction

Quiet submarine threats and high clutter in the littoral undersea environment demand that higher-gain acoustic sensors be deployed for undersea surveillance. This trend requires the implementation of high-element-count sonar arrays and leads to a corresponding increase in data rate and the associated signal processing. The U.S. Navy is developing a series of low-cost, disposable, battery-powered, and rapidly-deployable sonar arrays for undersea surveillance. The algorithms being mapped to these and other sonar arrays are computationally intensive, particularly when environmentally adaptive for detection and classification. As a result, the processing and dependability requirements placed on the data collector/processor, the monolithic embedded computer required to collect and process sensor data in a real-time fashion, are becoming prohibitive in terms of cost, electrical power, size, weight, etc.

Parallel processing techniques together with advanced networking and distributed computing technologies and architectures can be used to turn the telemetry nodes of these autonomous sonar arrays into processing nodes of a large distributed, parallel processing system for sonar signal processing. This approach holds the potential to eliminate the need for a centralized data collector/processor, reduce the aggregate battery drain, and increase overall system performance, dependability, and versatility.

The RDSA (Rapidly Deployable Sonar Array) architecture will be used as a baseline for purposes of comparison. The target current drain and cost for this array are 22 mA per telemetry node plus an estimated 2 A for the end processor and a cost of \$500 per node plus an estimated \$10K for a ruggedized, packaged end processor.

Products developed under this effort will include algorithms and performance models for the decomposition and mapping of signal processing applications to linear array or ring multicomputers for sonar arrays, a software-based fine-grain system simulator, a small-scale hardware prototype, and a prototype software system capable of running on both the simulator and the hardware prototype.

1.1. Technical Background

Large autonomous sonar arrays currently under development are designed along a basic "freight train" architecture. In this architecture the data taken at each node on the network is loaded onto a train which passes down the "track" to a centralized data collector/processor at the end of the track. The data collector/processor represents a single-point-of-failure for the network, a potential performance bottleneck, and is also a major cost driver. For example, a commercial VME-based special-purpose processor in a ruggedized package costs about \$10,000 and draws about 7 A of current. A future low-power version of this processor might draw as little as 2 A but this would still require on the order of 180 lithium C-cell batteries for a 30-day mission. The present program grew out of an attempt to use advanced parallel and distributed processing techniques and the high bandwidth and low latency of fiber optics to eliminate the centralized data collector/processor and replace it with a processing architecture in which each telemetry node of the network represents a processing element of a parallel processor, essentially turning the array itself into a distributed signal processing machine.

1.2. Technical Issues

The technical issues involved with the design and development of parallel and distributed computing architectures and algorithms for fault-tolerant sonar arrays include unsolved problems

in four major areas. These interrelated areas are: Topology, Architecture, and Protocol; Algorithm Decomposition, Partitioning, and Mapping; Fault Tolerance; and Simulator Tools.

1.2.1. Topology, Architecture, and Protocol

Given that all technical issues are driven by the network topology and node, processor, and network protocol design, the development of an efficient and effective architecture and set of protocols for this network-based multicomputer system for large sonar arrays represents a number of key technical challenges. Many important multicomputer architecture considerations must be addressed in terms of speed, cost, weight, power, and reliability for each node. These include the interconnect architecture, processor architecture, memory architecture, I/O architecture, and the proper hardware selection of components associated with each of these critical elements. Architectural components include circuits and devices for sampling, filtering, processing, communication, flow control, and clock recovery, and their design or selection poses a number of unresolved problems to balance performance and reliability with power, weight, size, and cost.

Network topology is an issue because it is a major cost driver and impacts both the algorithm decomposition and the protocol. The network may be linear and unidirectional, linear and bidirectional, ring-like or a hybrid combination of these. In a bidirectional topology data is passed in two directions, either over separate fibers or through the same fiber over different wavelengths. In a ring topology the last node in the array is connected to the first to form a ring-like structure. The choice of topology is strongly related to the cost and power requirements of the networking components used, in particular the optical transmitter, receiver, and clock recovery. A linear unidirectional network requires the smallest number of components but cannot support many parallel algorithms and does not support a high degree of fault tolerance. A linear bidirectional network is extremely robust and will support any parallel algorithm but requires twice the number of optical link components and a more complex protocol. The ring topology supports all algorithms but requires a long cable run and is the least fault tolerant. Newly developed plastic fiber represents an economical alternative to glass but has high attenuation, and therefore will not support a physically large ring. The choice of a topology requires a trade study of both the component characteristics and protocol and algorithm impact.

Network protocol is another key issue because currently available protocols are prohibitively power hungry. For example, a typical commercial FDDI chipset draws 400 mA. A custom protocol must be extremely efficient yet support real-time performance. A number of unresolved problems must be addressed include message-routing primitives and schemes, network flow control strategies, deadlock avoidance, and virtual channels. For all these unresolved problems in topology, architecture, and protocol, success is measured in terms of hardware and protocol flexibility, versatility, expandability, scalability, cost, weight, power, size, etc. as compared to the baseline ARDT sonar array.

1.2.2. Algorithm Decomposition, Partitioning, and Mapping

The design and development of novel parallel and distributed algorithms for large sonar array beamforming applications, and the partitioning and mapping of these algorithms onto candidate network topologies, architectures, and protocols, is one of the most pivotal technical issues in this project. This area is not addressed by conventional literature in the field. Rapidly evolving theoretical and algorithm developments in the area of digital signal processing will be the basis for program decomposition onto candidate multicomputer architectures for sonar arrays. These arrays may potentially exhibit multibeam and multifrequency operation, increased acoustic sensitivity, digital beamforming, and adaptive processing. When designing and developing

multicomputer architectures and protocols, the degree of potential matching between the architecture and the application algorithm must be analyzed via scalability studies. For different architecture and algorithm pairs, the analysis may often lead to very different conclusions. Therefore, another critical technical issue is scalability studies for the determination of the most optimal combinations of the candidate architectures with the most time-critical algorithms partitioned in various ways. Key problems in program partitioning and scheduling for distributed, parallel computers include granularity, latency, grain packing, and scheduling. Some of the parameters involved in this process include machine size (i.e. the number of nodes), clock rate or machine cycle, problem size, parallel execution time in terms of problem and machine size, I/O demand, memory capacity, communication overhead, cost, weight, power, size, etc. A number of decomposition techniques must be considered, including domain decomposition, control decomposition, object decomposition, and layer decomposition techniques, and each involves message-passing programming and performance tuning. Success is measured in terms of algorithm and software complexity, distributed/parallel performance, architectural correlation, flexibility, versatility, expandability, scalability, maintainability, etc. as compared to the baseline sequential sonar array.

1.2.3. Fault Tolerance

Another critical technical issue that must be addressed for this project is the determination of the most effective and yet efficient architecture and self-healing algorithm design strategies to improve the reliability and mission time of a large sonar array system. Like parallel and distributed algorithms for large sonar array applications, conventional literature in fault-tolerant computing does not address the unique requirements posed for these types of systems. Given the environment in which these systems will operate, component failure during the desired mission time is a major concern, repair is not an option, and it is imperative that the loss of individual processors, memory units, I/O units, interface circuits, and network links does not impede the ability of the system to perform with graceful degradation. A key element of the fault tolerance design studies conducted must be how best to improve fault coverage, reliability, and mission time while keeping low the power, weight, and cost factors. This balance is a critical unresolved problem for this or any system. Success is measured in terms of system reliability, mission time, graceful degradation, and the price paid for these in terms of power, weight, size, and cost as compared to the baseline sequential sonar array.

1.2.4. Simulator Tools

Given the increasingly high costs associated with the development of system prototypes, especially those based on complex systems designed for distributed and parallel processing, modeling and simulation techniques and tools have rapidly become a critical enabling technology and from them has arisen the field of rapid virtual prototyping. Based on Monte Carlo, Markov, Petri net, and other techniques, such simulation-based prototyping methods hold the promise to revolutionize the way in which computing systems such as those proposed for large sonar arrays can be efficiently and effectively designed. However, given the lack of appropriate simulator tools for the algorithms and architectures associated with this effort, another key technical issue is the development and exploitation of new simulator tools. For example, while no fine-grain modeling and simulation tools exist which alone are suitable for sonar array multicomputer architecture development, protocol development, fault tolerance, and program and algorithm decomposition and mapping, there are several tools which may be leveraged, adapted, developed, and integrated via simulator tools developed by this project and the optimum methods for this are themselves key technical issues in terms of both performance and dependability analysis.

Success is measured in terms of fidelity, flexibility, versatility, expandability, scalability, transition potential, and the simulator's inherent ability to demonstrate feasibility and assist in quantifying performance and dependability improvements versus sequential and candidate baselines.

1.3. Technical Approach

The technical issues will be addressed by a three-phase approach to the design and development of parallel and distributed architectures and algorithms for fault-tolerant sonar arrays. Together, these phases will work to demonstrate how advanced techniques in parallel and distributed processing, computer networks, and fault-tolerant computing can be effectively and efficiently employed to construct next-generation sonar arrays with less cost and size and a greater degree of dependability, performance, and versatility.

In the first phase, tasks concentrated on the study, design, and analysis of the fundamental components for these advanced sonar arrays. An interactive investigation of the interdependent areas of topology, architecture, protocol, and algorithms has been conducted. The results of this investigation include the optimum candidate network topology, architecture, hardware components, and interface protocols for the system architecture and a set of fundamental decomposition techniques and parallel algorithms for a representative set of time-domain and frequency-domain beamforming methods.

In the second phase, tasks concentrate on the critical steps to bridge from the results of fundamental studies in the first phase to the eventual goal of a small hardware prototype. Development has progressed on the preliminary software system for the advanced sonar array, the design of a strawman node circuit, and the development of a suite of simulation tools which support the design and analysis of both system performance and dependability. Parallel programs are being developed with inherent granularity knobs based on the results of algorithm decomposition and partitioning in the first phase. Self-healing extensions of the selected algorithms will be developed and simulated in this phase. A strawman hardware node design is necessary to determine estimated gate counts and power requirements. Finally and concurrently, a suite of simulation tools must be augmented and integrated to support the rapid virtual prototyping of topology, architecture, protocol, and algorithm selections made in the first phase.

In the third phase, the emphasis will be on the development, implementation, and demonstration of a small, laboratory-based hardware prototype with its own software system which will be fabricated, used to verify and validate the simulation results, and in so doing demonstrate and better quantify the inherent advantages of the novel approach employed in its design. Critical factors in the evaluation of the system will center on quantitative measurements in the areas of performance and dependability, including computational speed, efficiency, and precision, reliability, cost, weight, power, size, and mission time, as well as qualitative measurements related to system flexibility, versatility, expandability, scalability, etc. All of the measurements will be compared with the baseline ARDT freight train system architecture and sequential beamforming algorithms to measure the degree of success.

1.4. Summary of Results

In FY96 progress was made in several areas. First, an analysis of the effect of node outage on network reliability was conducted. This analysis indicated that network reliability could be greatly enhanced by using an optical bypass switch capable of bypassing at least two successive failed nodes. A switch of this type is currently being developed under the SBIR program. Second, a survey of low-cost, low-power networking components was started. So far laser diodes,

RAMs, ADCs, and high-power batteries have been investigated and analyses of low-power clock recovery circuits and plastic fiber optic cable is currently underway. When completed this survey will be used to determine the optimal network topology and system architecture. Third, preliminary parallel decompositions of several standard beamforming algorithms have been performed, including delay-and-sum, delay-and-sum with interpolation, and FFT. Algorithms and programs for the sequential versions of these beamforming techniques have been developed in Matlab, C, and MPI to form a baseline by which parallel algorithms and software will be measured. Fourth, a fine-grain model of the baseline freight train protocol has been developed using the Block-Oriented Network Simulator (BONeS) tool, and models for candidate network architectures are under development for unidirectional ring and bidirectional linear array topologies.

In FY97 further progress has been made in a number of areas. A taxonomy of decomposition and parallelization methods for conventional beamforming, both time- and frequency-domain, has been developed. Frequency-domain parallel beamforming algorithms using iteration- and steering-decomposition methods have been designed, developed, and analyzed. By coding these parallel algorithms in MPI as part of the preliminary software system, performance experiments have been conducted on a cluster testbed via several simulated network candidates. Results indicate near-linear speedup on both ring and bidirectional array networks and this speedup is critical since it will permit reduction in node and network clock rates, thereby further reducing power consumption. Time-domain parallel beamforming algorithms using iteration pipelining, transpose methods, and global data scope extensions are nearing completion, and early indications are also promising. Fine-grain array network models have been designed, developed, and verified including unidirectional, bidirectional, token ring, and register-insertion ring protocols, and a slotted-ring model is near completion. Medium-grain array node models are also under development and these models will permit parameterization and experimentation with clock speed, precision, operational versus standby power mode, etc. Integrating these many developments together, a new modeling and simulation environment for rapid virtual prototyping of advanced sonar arrays, called the Integrated Array Simulation Environment (IASE), has been designed and a working version is nearing completion. For the first time, IASE brings together the fine-grain network models, parallel beamforming software, and the medium-grain node architecture models in a manner that permits detailed experimentation with candidate algorithms, network architectures, and node architectures in a dynamic and integrated fashion. The evaluation of low-power networking and processing components was continued. A design for a low-power ($< 4\text{mA}$) optical link was developed and evaluated, and a 5 mA prototype using a commercial laser was fabricated and tested. Analysis of a lower-power version based on heterojunction bipolar transistor (HBT) technology was started and the development of a low-power clock recovery circuit was begun. The effects of component failures was also continued and simple and efficient techniques for sidelobe restoration in the event of node failure were evaluated.

1.5. Tasks

Task 1. Topology, Architecture, and Protocol Development

An analysis and selection/development of a candidate network topology, architecture, and interface protocol will be conducted. Hardware will include optical transceiver components, clock recovery circuits, batteries, protocol ASICs, and node processing elements. Low-power technologies such as HBTs will also be investigated. A high-level cost and power model will be

developed based on an analysis of fiber run lengths, approximate gate counts, and low-power optics.

Task 2. Algorithm Development and Modeling

A selection/development, analysis, and decomposition of digital signal processing algorithms with respect to their suitability in ring or linear-array multicomputer architectures for sonar arrays will be performed and an analysis and selection of parallel programming tools will be carried out. Representative algorithms from both time-domain and frequency-domain beamforming will be modeled and analyzed using analytical and experimental techniques so that tradeoff analyses can be conducted and the selection of algorithm and architecture features can be made. Modeling and analysis will be performed with respect to decomposition method and degree of granularity.

Task 3. Simulator Development

The results of the modeling and development process will be used to construct a fine-grain, high-fidelity simulator suite capable of accurately rendering node, network, and system behavior. This simulator suite will represent the performance of the basic freight train system as the baseline as well as the new distributed, fault-tolerant architecture and parallel, self-healing programs which are the emphasis of this project. The simulator suite will be equipped with a graphical user interface and will make it possible to gauge the potential performance of candidate architectures and algorithms and measure relative success and failure to achieve specific goals. The simulator suite will be expandable to include both low-performance and high-performance architectures for different mission requirements and will serve as the platform on which the preliminary decomposition, mapping, and tuning of the prototype software system will be developed.

Task 4. Preliminary Software System Development

Based on the results of algorithm decomposition and partitioning, a series of preliminary parallel programs will be constructed with inherent granularity knobs and portability for mapping to various distributed/parallel architectures. These algorithms and programs will be mapped and tuned for the linear array and ring topologies under consideration. Parallel programs shall be extended to include basic self-healing mechanisms. A series of test and evaluation experiments will be conducted in order to determine the tradeoffs implied by the matching of these parallel architectures and algorithms.

Task 5. Strawman Node Circuit Design

A strawman functional circuit design will be conducted and a high-level gate count and node power requirements will be determined based on a COTS and ASIC foundry analysis and power consumption evaluation. A functional processor/interface circuit will also be designed. The results of the simulations together with the strawman design will be used to verify that system performance, fault tolerance, and power requirements are met, and the process will be iterated, if necessary.

Task 6. Algorithm and Preliminary Software Development for Split-Aperture Beamforming with Cross-Spectral Correlation

By extending developments in Tasks 1-4, modeling and tradeoff analysis of decomposition methods and degrees of granularity will be conducted for split-aperture beamforming algorithms with cross-spectral correlation. New parallel algorithms will be developed and studied via basic performance models, and preliminary software will be designed for mapping, tuning, and evaluation.

Task 7. Hardware Prototype Development

An operational small-scale hardware prototype of the network will be constructed that will support the detailed decomposition, mapping, and tuning of the prototype software system and verify simulation results.

Task 8. Detailed Software System Development

The preliminary parallel and self-healing programs will be extended to form the development of a prototype software system for this distributed signal processing machine. These programs will be mapped to the hardware prototype, tuned for performance and dependability, and a series of test and evaluation experiments will be conducted to ascertain and demonstrate the superior capabilities of this hardware and software system as compared to their baselines.

2. Integrated Simulation Environment

The Integrated Simulation Environment (ISE) was begun as a response to the growing need for a system that simulated (or emulated) a parallel multicomputer from the interconnection network (ICN) to the actual parallel application. The ISE may be used to model any node architecture, software protocol, O/S overhead, and fine-grained network. Existing methods to measure simulated performance do not support data patterns of real-world applications. Furthermore, existing integrated solutions do not provide the fine-grain detail of stand-alone nonintegrated network models. ISE resolves these problems by incorporating high-fidelity models of complex network systems that are capable of supporting real data.

The ISE profits from the ability to build a parallel application, a node architecture, and an ICN with any desired level of fidelity within a commercial modeling tool. The heart of this system is an application program interface (API) called BONEs/MPI. BONEs/MPI is a communication protocol between BONEs and real MPI processes that allows these processes to offer real-world traffic to the node architecture and network models.

The Integrated Array Simulation Environment (IASE) benefits from such a powerful tool. The theoretical array is a linearly distributed set of sonar nodes that perform the same MPI beamforming applications debugged on a laboratory testbed. The IASE is helping to build efficient real-time parallel beamformers without having to purchase and test actual hardware. Besides providing a dollar savings, the ISE allows systems to be rapidly prototyped by swapping integrated models. For instance, a register-insertion ring can be swapped for a slotted ring while maintaining the same node architecture, processor model, and beamform application. The ISE has the potential to one day be a tool for rapid virtual prototyping of any system with any desired fidelity for savings of time and money.

The following subsections will present the parts and processes of the ISE. This presentation begins with an overview of the suite of programs and structures comprising the ISE. The remainder of the section describes major components incorporated into ISE: parallel program interfacing, microprocessor modeling, and network modeling.

2.1. Foundation Components

The Block-Oriented Network Simulator (BONEs), a commercial program created by the Alta Group, is the discrete-event-driven simulation package used for all networking modeling, simulation, and analysis in this project. Its main features allow one to build a data structure (Data Structure Editor), operate on the data structures with hierarchical process blocks (Block Diagram Editor), execute the block models (Simulation Manager), and build output graphs (Post Processor). The term *event-driven* refers to simulations that sequentially execute through a dynamic list of events, which are simple descriptions of every simulated action. Events are propagated through the network of structures in time slices. Events do not actually take up time until they encounter a delay block; therefore, many events can happen in the same time slice, thus giving the appearance of simultaneous occurrences. The execution of the simulation randomly traces events that occur in zero time until all events of the slice of time are consumed. The clock is then incremented, and the next set of events are executed based on a set of scheduling rules. This process is continued until the entire simulation time is completed. Not all events are completed in random sequence. Some blocks in BONEs give the user execution controls to ensure that certain events occur in a determined sequence when within the same time-slice. BONEs simulations are completely fabricated in C/C++ code from blocks of code called primitives. BONEs designers can use this capability to their advantage in creating new primitives

and are limited only by C/C++ and the operating system. In fact, the code necessary to run the ISE has been created in these primitive blocks.

The Message Passing Interface (MPI) is the foundation on which the parallel programs for the ISE are built. MPI is a standard specification for processes communicating via messages [MPIF93]. In the message-passing paradigm, each process has an identification number. To send information to another process, the identification number of the destination process is needed. Communication between processes cannot occur except by explicitly calling functions to connect with the other nodes.

The message-passing paradigm fits the sonar array project well due to the fact that each array node will have its own local memory. Message passing is the preferred method to use when the multicomputer is loosely coupled. Since the sonar array is comprised of distributed, autonomous nodes with independent processors and memory, the use of a multicomputer message-passing software package such as one compliant with the MPI standard is essential to the project.

Of the hundreds of defined calls in MPI, there are a number of basic cornerstones which have been implemented in the ISE. These calls include standard send, standard receive, broadcast, barrier, probe, non-blocking probe, and reduction (with various operations), among others.

2.2. Structure of Environment

The ISE environment is comprised of several entities, each serving a specific role in the communication of data and timing information between the BONEs network and the user parallel processes. The structure of the environment is shown in Figure 2.1. More detailed explanations of relevant portions of this structure are presented later in Appendix A.

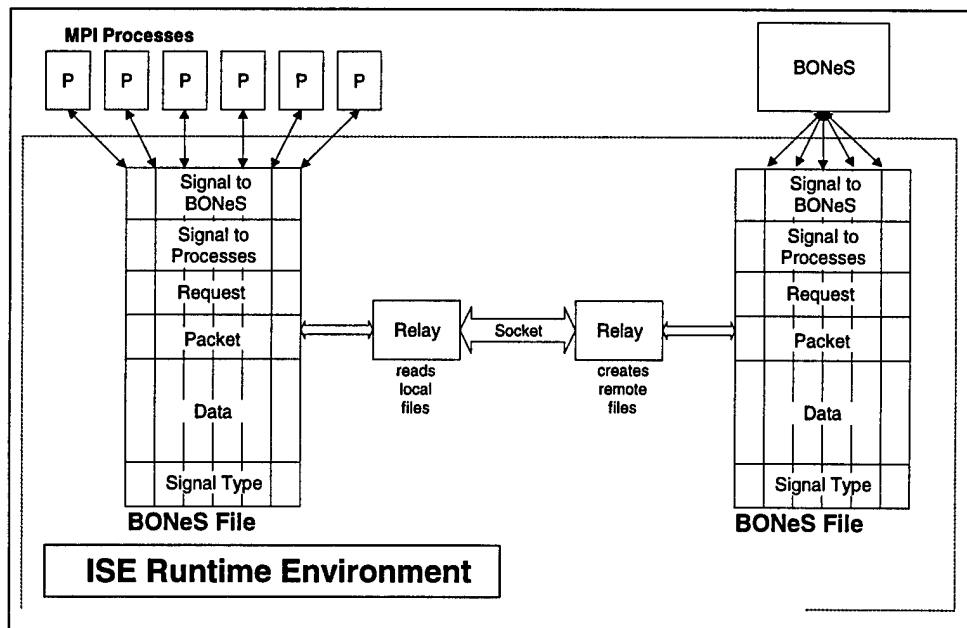


Figure 2.1 – ISE Runtime Environment

2.2.1. Parallel Process Code

The parallel programs are written in the standard MPI-C parallel coordination language [MESS94]. Instead of providing routines to access hardware, the ISE version of MPI interfaces with shared files. All communication to and from the parallel processes is made through these shared files, which is accessed by the ISE runtime system. The access is completely invisible to the MPI programmer.

The user MPI program can include the following basic MPI calls as well as the ISE-specific *MPI_Barrier_time* and special non-timed versions of the regular MPI calls. Note that these extra non-compliant calls are extensions to the standard MPI specification and will not interfere with program operation. They will, however, be unrecognized in any other MPI implementation.

The standard *MPI_Send* call allows a user to send a specified amount of information from the user memory space to a specific remote workstation. The sent message includes a tag value, which is used to distinguish the message from other messages with the same source and destination. How the user program utilizes this tag is dependent on the application and is transparent to the MPI implementation. The *MPI_Send* call returns control to the user program once the data has been copied out of the user space. This call may or may not block the user process for some length of time. If the ISE application layer in the network model has sufficient buffer space, the call will return immediately. If the application layer is out of buffer space (or for any other network-specific reason), the *MPI_Send* call will block until the application layer is free. Note that in either case, the communication is called “non-blocking” since the call returns without waiting for the destination node to receive the message.

The complement of the *MPI_Send* call is *MPI_Recv*. A process wishing to receive data with a specific tag from a specific node uses this call to block until that data has arrived and has been safely copied to the desired point in the user space. The user may specify the “ANY_SOURCE” wildcard to indicate to the MPI implementation that it should return a message from any originator as long as the tag matches. The user can also specify the “ANY_TAG” wildcard to accept all tags. Whenever a wildcard is used to receive, a status variable returned after the call indicates the tag and source of the message actually passed to the user.

The probe call, *MPI_Probe*, operates the same way as the *MPI_Recv* except that the data is not copied out of the implementation buffers into the user space. Instead, only the status structure is returned. The non-blocking version of this call, *MPI_Iprobe*, returns a flag that indicates whether or not a matching message has been received. Rather than block until the message arrives, the *MPI_Iprobe* can be used to poll the network and return without a match. If a match was successful, the user program may use the status variable returned from the probe to issue an *MPI_Recv* on that message.

In the broadcast call, *MPI_Bcast*, all processes make the same call with the same arguments. One of the arguments is the source of the data. When the *MPI_Bcast* is called and the source is equal to the rank of the calling node, the MPI implementation realizes that the source is itself and does not receive. If the source in the call is not equal to the node rank, the MPI implementation will receive the broadcast. The sending side of the broadcast may be implemented by a series of individual sends to all other nodes, or it may take advantage of hardware optimizations for broadcast by sending to destination node “-1.”

The *MPI_Barrier* call is used to block the processes until all processes have reached the call. In the ISE, this function is implemented with a single-phase all-to-all transmission. Once each MPI node has received a token message from all the others, it knows that all the other processes have reached the barrier, thus the barrier is passed, and the call returns.

The reduction operations with *MPI_Reduce* take as input a vector of some length from all the nodes. These vectors will arrive at a destination node and be reduced according to some rule. For example, the rule may be to find the element-by-element sum, maximum, or minimum of all the nodes' vectors. The *MPI_Allreduce* function carries out the same operations except that after the call is finished, all nodes have the vector result instead of just a single destination node.

The last formal MPI function included in the ISE is *MPI_Wtime*, which returns the current time. Since the network simulation in BONeS runs independently of the MPI parallel processes, the *MPI_Wtime* included in the ISE is the only way for the processes to read "wall clock" time. If they use the standard "gettimeofday" or other UNIX clock functions, the values returned will be those for real-world time, which will be incorrect because BONeS is keeping the time in the simulation.

In addition to these standard MPI calls, the ISE provides a number of additional calls meant to ease the burden of simulation or to provide additional features. The *MPI_Barrier_time* call forces the process to block until some specified absolute simulation time. This functionality is in contrast to *MPI_Barrier* which blocks a process until all processes have reached the barrier. Examples of the use of *MPI_Barrier_time* include blocking a processes until an analog-to-digital converter has supposedly placed data into the process memory, as is done between iterations on the beamforming simulations.

All processes are divided into code blocks, which are defined as all the non-parallel code located between successive MPI calls. On entrance and exit from any MPI call, the wall clock time on the system will be noted. The execution time of a code block is determined from subtracting the entrance time from the exit time of the previous MPI call. The ISE uses these code block times to know at what simulation time the next request from a process must be retrieved and serviced. Knowing these times beforehand and reading the request only at these times allows the ISE to service requests at the appropriate times without having to poll the processes on every clock cycle to see if they have requests, thus improving the speed and efficiency of the simulation. The ISE includes a parameter so that the actual code block time on the workstation can be scaled by a constant factor. This factor allows the user to simulate using processors that are faster or slower than the processors available for use. For example, if the user wants to estimate processors running at half the speed of the processor of the host computer, a factor of 0.5 can be specified.

As a result of using wall clock times, the execution time of a code block will include any time when the multitasking operating system has swapped the process out; therefore, the execution time of the code blocks reflects the loading of the workstation used for process simulation. It will also include time the process spends in system calls such as memory allocation. These results are major features of hardware-in-the-loop simulation; the hardware provides the simulation with real-life performance numbers.

All the parallel processes run on the same workstation, and thus are not actually running in parallel. The processes appear to the network simulation as operating in parallel because each process keeps the simulation time at which it makes an MPI call independent of the times for the other processes. By placing all processes on the same workstation, the ISE does not slow down the simulation time since the BONeS network simulation is considerably slower than the hardware-in-the-loop code-block execution. Because all processes are on the same workstation and can be swapped out during code block execution, a mutex (i.e. mutual exclusion device) is used between the processes. Each process must lock the mutex before timing its code block. This procedure prevents one process wanting to time a code block from preempting another process in the middle of timing a code block. The process wanting to time must wait until the process currently timing is done and releases the mutex. As such, the code block times may only

include unwanted preempting from non-ISE processes on the workstation that are not using the mutex.

The ISE provides a distinction between initializing MPI and *MPI_Init*. The *MPI_Init* call in the ISE begins the simulation; however, it is often desirable to include setup routines in the processes which use MPI but are not part of the desired simulation. In such a case, the user initializes the ISE with the *MPI_Notime_init* call. After this call, the processes carry out the setup routines. These routines may include MPI calls, but such calls are not simulated through the BONEs network. After the processes are done setting up the simulation and want to begin, they call *MPI_Init* to begin timing. Likewise, at the end of the simulation, the user may call *MPI_Stop_timing* to end BONEs's participation, then proceed with some cleanup routines before calling *MPI_Finalize*. The strategy of including only relevant code in the BONEs simulation while allowing setup and cleanup code to be non-timed speeds up the simulation process. To expand the functionality of the non-timed code, the ISE includes non-timed versions of all MPI calls. Communication can proceed using these calls outside of the network simulation.

2.2.2. MPI runtime system

The ISE runtime system is in charge of setting up the simulation, managing the shared file interface, and providing other support for the processes and BONEs. The following paragraphs describe the actions taken by the system when interfacing an MPI program to BONEs.

First, the runtime system creates the shared files necessary for communication with the parallel processes. The purpose of the shared files is to provide each process with a specific location to which it posts requests and outgoing data and from which it receives incoming data. When making a request or sending data, the process will place the data in the file and post a flag. The flag will be used to tell the runtime system that there is data in that node's share of the file. When receiving, the process will wait until a receive flag is posted before accessing the file and reading the data. Second, the runtime system creates the relay programs on the local host workstation and on the workstation which will be running BONEs. The purpose of the relay program on the local (parallel process) workstation is to poll the flags for all the processes. If there is a request, the relay program sends the data and the originating rank number through a socket to the remote relay program. At the remote relay, the rank number and data are read from the socket. The data is placed in the appropriate location for the rank, then the flag for that rank is posted. At this point, the BONEs node that corresponds to the rank will be able to read the data. The same procedure occurs in the opposite direction when a BONEs node sends data to a parallel process.

Next, the runtime system spawns all the parallel processes on the host workstation. To spawn the processes, the runtime system places a unique rank number in a well-known location in the shared file and spawns one of the processes, which immediately looks up its node rank in the shared file to place in a local variable for later identification. Once this is accomplished, the process posts an inter-process semaphore on which the runtime system has been waiting. The process then attempts to lock the mutex and time its first code block. After the runtime system has successfully waited on this semaphore, it overwrites the shared file with the next node rank number and spawns the same process again. The process will then identify itself as a different rank and executes accordingly. Once this procedure is complete, all processes will be running and timing their code blocks.

During execution, each process must notify BONEs of its next scheduled MPI request. The parallel process adds the execution time of the code block preceding the MPI call to the current simulation time, which it received from BONEs at the end of its last MPI call. The process then sends this request type and time to BONEs by writing them to the local shared file and posting the

flag. The local relay will discover the flag has been posted and write the data to the remote relay through the socket. The remote relay will read from the socket, write the request type and time to the remote shared file in the column for the requesting node, and post the flag for that node's column. BONEs will then recognize that there is a request from that node. BONEs must always contain a request for each process so that it knows at what time the process needs a service. To make sure it always has a request, BONEs blocks on reading a new request after servicing the previous request (before proceeding through any more simulation time steps). Blocking on reading a request does not increment the simulation time. This procedure ensures that BONEs will not get ahead of the processes unless BONEs is servicing a previous request.

After sending the request, the process writes any information BONEs needs to execute the request to the local shared file. The process then needs a reply, so it will block until the flag in its column of the local shared file is posted by the local relay, meaning the reply has been written by the local relay from information it has received through the socket from BONEs. The reply may include data for a receive request. The reply also contains the time at which the request is satisfied so that the process can update its simulation time. Once the process gets confirmation of completion of the request, it repeats the procedure by trying to lock the mutex so that it can time its next code block and then make its next request.

After reading this description, it is important to note that the above operations are all implemented within MPI calls. The MPI calls appear to the user to be exactly the same calls as used in major MPI implementations, thus the interface to the ISE is made completely transparent. The user need not know how the underlying mechanism is working, only that it executes the expected way. Because of this transparency, the ISE is a powerful prototyping tool that supports the portability of software (algorithms) developed by the user.

2.2.3. ISE Version 1.3 Extensions

Several extensions to the ISE were completed in recent weeks. These improvements have no bearing on the explanation of the ISE operation above. They add considerable flexibility to the ISE so that the user may more conveniently run simulations.

2.2.3.1. Multiple Iterations

The ISE is designed to exploit BONEs's ability to spawn multiple simulations (also called iterations) on different machines by specifying multi-valued parameters. Since each single-iteration simulation runs all of its parallel processes on a single machine (a different machine than the one on which the BONEs simulation is running), a simulation with x iterations requires x machines dedicated to the processes and the timing of the code blocks in those processes. Also, x machines can be used to run the BONEs simulations. In such a configuration, all x iterations can be completed in approximately the time it takes for a single-iteration simulation (namely, the single-iteration simulation for which the parameters cause the longest simulation time).

BONEs is completely flexible when assigning machines to iterations insofar as BONEs will assign an iteration to a machine when available. If a machine is not available or the user has not specified enough machines for all iterations, BONEs will assign what iterations it can and simply assign the remaining iterations as the initial iterations complete. Furthermore, BONEs is free to decide for itself the mapping of iterations to machines. That is, the order of the machines specified by the user is unrelated to iteration numbers. The ISE supports this flexibility in BONEs with an appropriately flexible interface between the BONEs machines and the process machines. The ISE is, however, constrained in that the processes for an iteration must be matched correctly with the BONEs iteration no matter where BONEs places that iteration. When

the BONEs simulation for a particular iteration begins, an initialization routine creates the remote relay program on whichever machine the BONEs iteration is running. Due to the fact that the type of machine on which the processes are run (whether it be a SPARCstation-5, SPARCstation-20, UltraSPARC, etc) is important, the process machines are fixed to iteration numbers at runtime. A file is created for each iteration (the bridge file), and the name of the local (process) machine for the iteration is placed in the iteration's file. The file is called the bridge file because it contains static information about an iteration and can be read across multiple machines without coherency errors, thus bridging the machines without the need for a dynamic relay as is needed for the communication shared files. The local relay is then spawned, finds a valid socket port, and waits for a connection. The remote relay is told to which iteration it belongs by the BONEs iteration (via a "Get Iteration Number" block in BONEs). The remote relay can then look for the bridge file for its iteration, find the process machine for its iteration, find the port to use, and connect to the local relay of the correct iteration. The iteration is then ready to run and proceeds in the same manner as the single-iteration ISE previously described.

2.2.3.2. Runtime System

A runtime system for starting ISE simulations was created to help in setting up the parameters and spawning the processes from the different iterations on the appropriate machines. The system can either read a configuration file and begin the simulation without user intervention or it can prompt the user for all necessary information. In the latter case, the user is asked how many iterations to run and how many nodes (which must be the same for all iterations) are to be simulated. Next, the runtime system steps through each iteration, asking the user for the required information. First for each iteration, the system asks the user which class of machine the iteration is to use as its hardware in the loop. Next, the system asks which specific machine in that class is to be used and displays the processor utilization of that machine. If the machine is heavily loaded with processes from other users, the ISE user may elect to reject the machine and choose another. In that case, the next time the user chooses that class of machine, the runtime system lists the rejected machine as "rejected" so that the user remembers which machines were loaded. Once the user accepts a machine, the system asks for the command line of the parallel program and the processor performance-scaling factor (i.e. 0.5 to simulate a machine with half the processing power of the chosen real-world machine). The system also asks the user whether to record profiling information so that Upshot, a freeware graphical profile viewing program, can be used at the end of the simulation to graphically view the communication of the iteration [HERR91].

The system then proceeds to ask the user for the required information for any additional iterations. However, in later iterations, the process machines which have already been assigned to iterations are "grayed out" and cannot be chosen. This mechanism ensures that the processes from different iterations are on different machines and do not interfere with each other's code block timings. Once all iterations have been configured, the runtime system allows the user to save the information in a configuration file. This file can then be used straight from the ISE command line to avoid the above described prompts. The ISE then creates an output window for each iteration and asks the user to start the BONEs simulation with the given number of iterations and nodes.

Once the multiple-iteration simulation is complete, each window will prompt the user to display the graphical profile. If activated, the ISE spawns Upshot, which automatically brings up the corresponding profiling log file. At this point, the iteration window prompts the user to hit a key to end. Until the user hits a key, the window remains open so that the user can read the program output.

One last feature of the ISE runtime system is that it checks for currently running simulations and/or improperly terminated simulations for the user. The reason it must check this condition is that every ISE execution starts numbering its iterations from zero. When it spawns iteration zero, any shared files for a currently running simulation or a terminated simulation will impede creation of the shared files necessary for the new iteration. Therefore, the ISE checks for a lock. If it detects the lock, it informs the user that the lock is present and that there is either a simulation running or there had been an improperly terminated simulation. A cleanup routine is provided so that the user may remove the lock if it is known that the lock is not due to a currently running simulation.

2.2.3.3. Trigger Network

The MPI function set for the ISE was expanded to allow additional capability to the MPI programmer when using the ISE. The function *MPI_Trigger_network* allows the MPI programmer to specify at a certain point in the program that a trigger should be engaged in the BONEs model itself. There are three general-purpose triggers (specified by the argument to the function) available out of the BONEs/MPI Portal block to which the BONEs modeler can connect. This mechanism gives the user the ability from the parallel program to create a BONEs event to trigger any BONEs block. Doing so requires knowledge of which trigger number corresponds to which event; however, the additional capability may be well worth the cost. For example, the user may wish to run a portion of code, report timing results, trigger a memory parameter change in BONEs (such as network speed), then run the same code or additional code over the newly configured BONEs model.

2.3. 'C54 Simulator

To better understand the complex processor architecture of the prototype, an in-house 'C54 simulator was written in C++ (in addition to the proprietary Texas Instruments simulator) and used to execute the sequential algorithms already written. The in-house simulator reads an assembled file in S-record format, puts the program in program memory, and executes the instructions in the same way as would the actual processor. The simulator reports program and data memory contents, register (including the program counter and stack pointer) and accumulator contents, and number of clock cycles per program executed. The simulator is still under development with the goal of becoming a servicing part of the ISE. Several more instructions will be implemented, and additions such as a model of the TDM serial port on the 'C54 single-board computer are being added. A parallel simulator is being developed where the simulator will be implemented as an object-oriented class, and several processors will be instances of this class. The processor classes will communicate with each other through the simulated TDM port.

The final goal of this ongoing project is to create a BONEs primitive to be used in the ISE in order to have a more precise software prototyping environment. Such a system would present an alternative to the hardware-in-the-loop timing of MPI code blocks. Using the in-house 'C54 simulator as a block in BONEs enables us to accurately simulate and debug the behavior of the complete sonar array before it is actually implemented in hardware.

2.4. High-fidelity Network Models

This section presents the two high-fidelity, fine-grain network models created during the past year for use as ISE networks and possible DPSA implementation. The first model is for a slotted-

ring network. The second is a variation of the slotted ring called the ratchet network, which aims to provide a more robust network synchronization protocol. These two network models complement the networks completed in FY96 and described in our first Annual Report.

2.4.1. Slotted Ring

A new programming model, which we are calling a Global Data Scope (GDS) system, has changed the number of viable networks for the DPSA. GDS is a coarse-grained model that uses a technique popularized by VMIC (VME Microsystems International Corporation) called reflective memory. In this type of system, every node has a copy of the memory contents of the other nodes. These distributed memories allow global data to be accessible to each node. Because of this, each node does not have to worry about explicitly passing data to other nodes; only command information must be sent explicitly. Figure 2.2(a) below shows a high-level diagram of the GDS approach. Notice that in the GDS model only commands are issued from one beamforming layer to another. The more conventional approach may be called Local Data Scope (LDS). In an LDS system, the beamforming algorithm must not only issue commands to other nodes but must pass any data to those nodes which is required of any command. The LDS model is shown in Figure 2.2(b) below.

The advantages of a GDS system are many including program efficiency, fault-tolerance, and simple parallel decompositions. These advantages are discussed in further detail in subsequent sections. However, one of the best advantages is the large use of broadcast sends to the network. Broadcast sends tend to be very efficient if supported by the hardware.

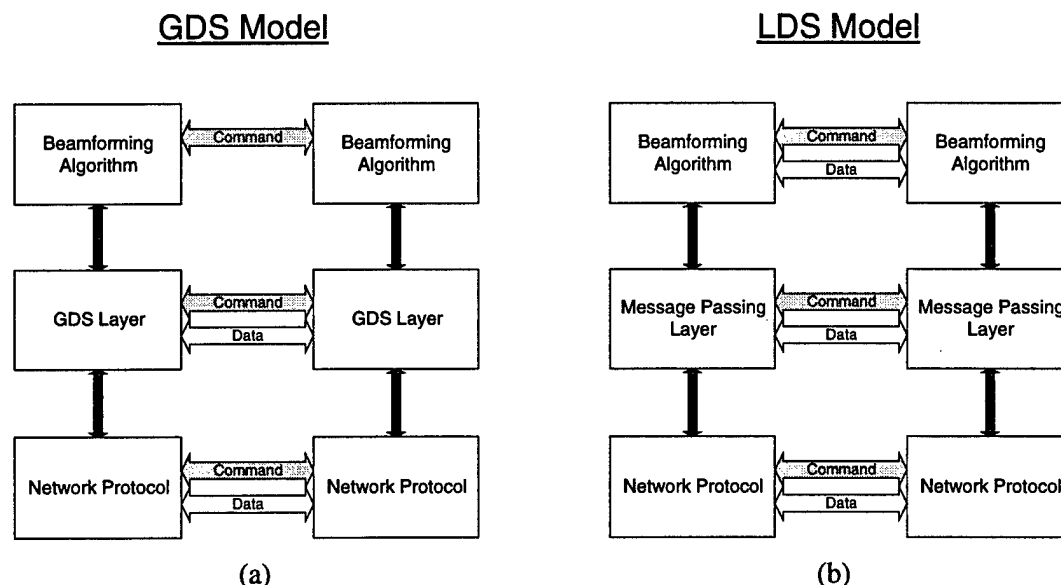


Figure 2.2 – Global Data Scope (GDS) versus Local Data Scope (LDS) Systems

The slotted ring was not initially considered to be a viable ring medium access control (MAC) because of the existence of superior contention schemes such as register insertion. However, it offers other distinct advantages such as synchronization. However, in the case of mostly broadcast traffic, it can approach the efficiency of register insertion rings. This protocol would not be optimal for many beamform applications, but for the Global Data Scope (GDS) system in particular, it is very well suited. This section discusses the functional aspects of the slotted ring protocol, verifies and validates the function of the protocol, and lastly compares the

performance of the protocol against a register-insertion ring and a register-insertion bidirectional array.

The slotted ring is not a well-defined protocol. It may be characterized as a reservation contention scheme and also round robin. Contention is provided by slots that rotate like a train running in a circle and connected at the extremes. The ring size and ring delays determine the number of slots, which is the sum of propagation delays and node delays. In other words, the slots are arranged so that they are continuously flowing through each node, implying that the architecture will be cut-through to some degree.

The rules are simple: a node may use any empty slot that passes as long as the node does not already have data on another slot. Once filled, the slot will propagate around the entire ring until returning to the origin. The node then strips off its data and sends an empty slot. The node may not reuse this particular slot again but must forward it to the next node and wait for the next empty slot to arrive. This small detail ensures fairness to all nodes. The contention protocol just happens to provide two services which are necessary for the DPSA: efficient broadcasting and tight synchronization. The protocol is obviously efficient at broadcasting since each packet traverses the entire ring and may be directed to each node's input along the way. Also, fixed-size slots (or packets) ensure tight, easy-to-implement synchronization.

The slotted ring MAC is based on evaluating and selecting one of three conditions for incoming packets: store-and-forward, sink-and-create-empty-slot, and forward. These three conditions completely characterize the movement of any slot through an input switch. The store-and-forward condition occurs when the packet is addressed for that node. The sink-and-create-empty-slot condition is the result of a node receiving a packet which originated from that node. The last case considers the situation in which a node receives a packet that neither is destined for it nor has originated from it. This simple input switch is shown below as Figure 2.3. It also allows a broadcast message by using the well-known broadcast address, "-1" (negative one).

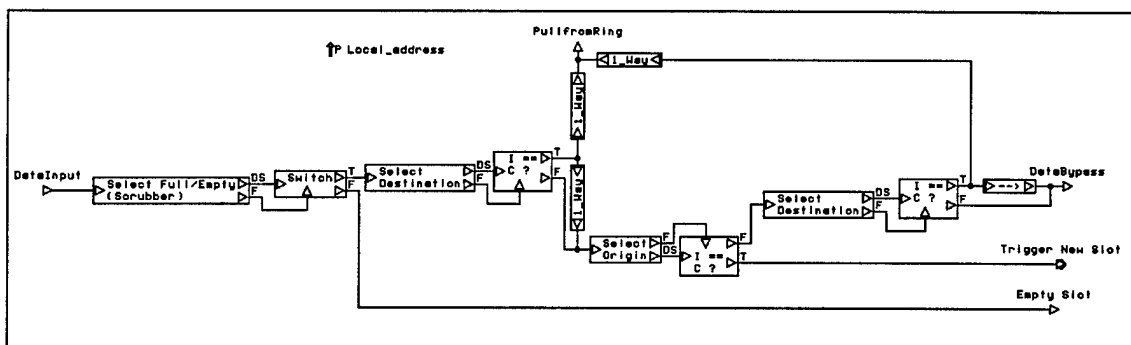


Figure 2.3 - Input Switch. The input switch performs checks on three conditions: if the incoming slot is destined for that node, if the incoming slot originated from that node, and if the slot is empty and may be fed to the output multiplexer.

The forwarded packet from the input switch then must also be queried before being output to the network. With certain conditions present, the node may have the opportunity to use this slot for its next transmission. The variables for this condition are: status of the slot (empty/full), status of the output queue of the node, and the transmitting status of the node. These conditions may be shown as a Boolean table such as in Figure 2.4 below. Clearly from this table, three conditions must be met for a node to use a slot for transmission. The slot must be empty. The node must have something to send. Lastly, the node must not be transmitting on another slot. The last condition helps to provide fairness but unfortunately also limits the performance of the network.

Boolean Table - Conditions for Writing to Passing Slot

Node's Ability to use Forwarded Slot
(condition - not currently transmitting)

Slot Status

	Empty	Full
Empty	Can't Use	Can't Use
Not Empty	<u>Can Use</u>	Can't Use

**Output
Queue
Status**

Node's Ability to use Forwarded Slot
(condition - currently transmitting)

Slot Status

	Empty	Full
Empty	Can't Use	Can't Use
Not Empty	Can't Use	Can't Use

**Output
Queue
Status**

Figure 2.4 - Slot Usage. Certain conditions must apply before a node may write its data onto a passing slot. These conditions are broken up into a Boolean table above.

The output decision process is conducted by the *Fill Empty Slot* module as shown below in Figure 2.5. The BONEs simulation routes full slots around this slot handler; therefore, it simply tests the two remaining conditions: whether the node is currently sending and if there is something in the queue to send.

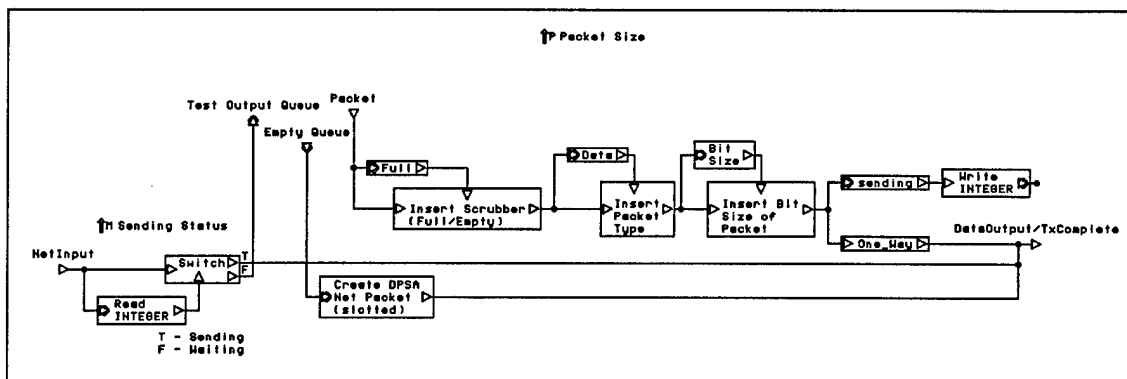


Figure 2.5 - Fill Empty Slot. This module attempts to fill passing slots with data if conditions are right. However, if the node is currently waiting on another transaction or has an empty output buffer, the slot will be transmitted empty.

2.4.2. Test Applications

An application layer was created to measure the performance differences between the register-insertion ring, the register-insertion bidirectional array, and the slotted ring MAC protocols. This layer requires only three parameters: the message size in bits, the hop count, and the message rate. The message size is a parameter, which can be set to arbitrary size. This parameter affects the efficiency of each transaction by adjusting the proportion of actual data bits to packet overhead. The slotted ring has the additional restriction of slot size that may further restrict its efficiency. The hop count is a parameter that may be set to any value (even larger than the number of nodes in the array). The application layer actually uses the hop count number and

its rank number to calculate the destination node (via taking the modulus of hop count plus node number against the number of nodes). For example, node 2 and a hop count of 4 will send to node 6; whereas, node 6 will send to node 2 in an 8-node array. The message rate is a parameter that is set in the saturation region causing data to fill the queues to capacity. This setting helps determine the maximum throughput during overload and also helps to determine the maximum latency before the saturation point.

The first tests were validation tests of the slotted ring for performance and proper message routing. A series of ISE tests were quite convenient to verify message routing. The “ping” test simply sends dummy messages (ping-pongs) between multiple nodes. The ping test is used as a simple tool to test the functionality of the network. The idea is to ensure that nodes can generate and respond to messages much like a “ping” request is done on real machines to test if they are alive. The ping test was successful and each node was able to receive a message from every other node. Various other tests were performed to check the functionality of the network including its ability to broadcast messages, barrier synchronize, probe incoming data, and terminate.

The second set of tests for the slotted ring validated the network’s performance. Two tests were performed to gain insight into the slotted ring’s characteristics. The first of these tests injected data at a uniform rate into the system past the saturation point. Multiple iterations were run changing the slot data size from 32 bits to 256 bits while keeping the size of the data packets injected into the system fixed at 256 bits. A *packetizer* block was needed to break up these larger blocks into the smaller data blocks. Intuitively, breaking a message into packets seems less efficient than sending the entire block. As is shown in Figure 2.6, this assumption is approximately correct; however, a seemingly strange phenomenon occurs when approaching the size of the packet. For instance, when sending 256-bit packages in 224-bit slots, the performance is poor. This result may be explained by observing that each packet almost fits into the single slot, but a whole additional slot must be used in order to complete the transaction. Also, note that this plot shows that the slotted ring is most efficient when it can break a packet into integral pieces of the whole. Thus, the most efficient slot size for the 256-bit packages is the 256-bit slot, followed by the 128-bit slot, followed by the 64-bit slot, etc. The worst efficiencies are observed when sending a slot that is slightly bigger than the packet size. Also, sending very large slots (compared to packets) is also inefficient. Slotted rings must be specially tuned to the expected size of network traffic to be efficient. However, since the DPSA has very deterministic traffic patterns, inefficient cases may be avoided.

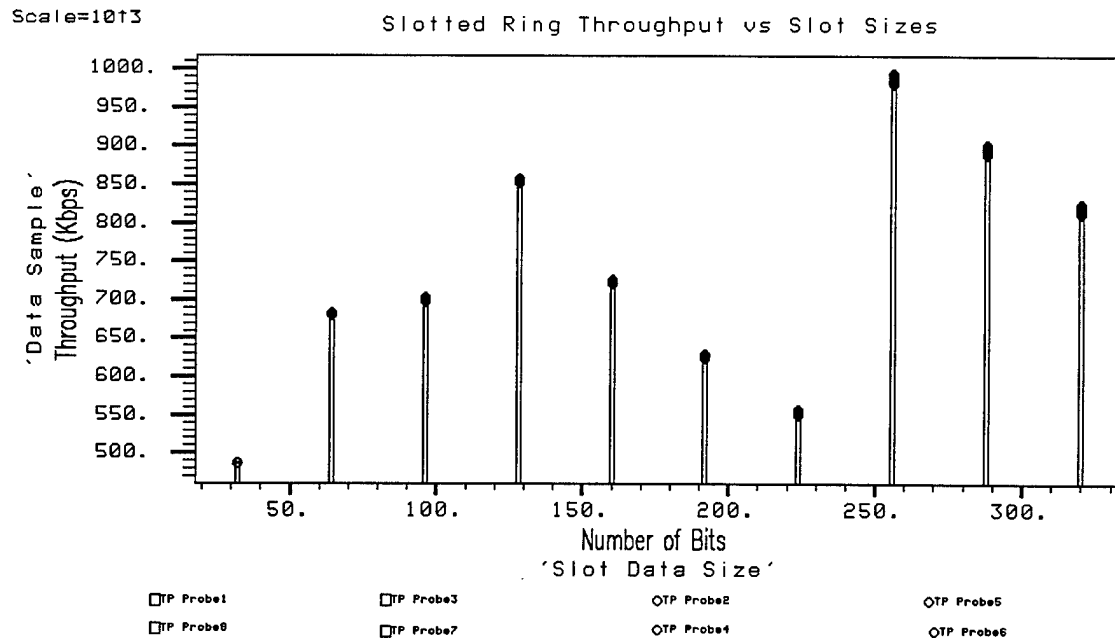


Figure 2.6 - Throughput versus Slot Sizes. This test shows the response of the throughput while fixing the size of input data packets to 256 bits and simulating the system over a range of slot sizes. Note that the network is most efficient when the incoming network packets equal the slot size.

The second test was a performance test versus the number of slots in a fixed eight-node system. Although the results are as expected, the plots do not reveal the total nature of the system. The number of slots should never be set to a value larger than the number of nodes for the contention scheme implemented. Recall that a node could only use a slot if it were not currently waiting on a previous transaction. If the number of slots is set to a number greater than the number of nodes, the system is guaranteed to always have a number of empty slots rotating around the ring wasting bandwidth. If a more complicated contention scheme was used, perhaps this situation might work, especially if the physical link lengths were large, indicating a large propagation delay. This test was also performed at saturation, which led to another result. That is, setting the number of slots equal to the number of nodes appears to be the ideal case (as shown in Figure 2.7). However, systems are typically not run at the saturation point and instead may only use a percentage of the total available bandwidth. It seems intuitive, again due to the contention scheme, that the number of slots should be designed for the average number of nodes requiring service. This number is somewhat related to the average load by each node. If the average load for the nodes is 80% of the saturation load, one can deduce that the average number of nodes requiring service at any moment is 80% of the total number of nodes. Therefore, an eight-node array might be most efficient with 6 slots if designed to operate 20% below the saturation point.

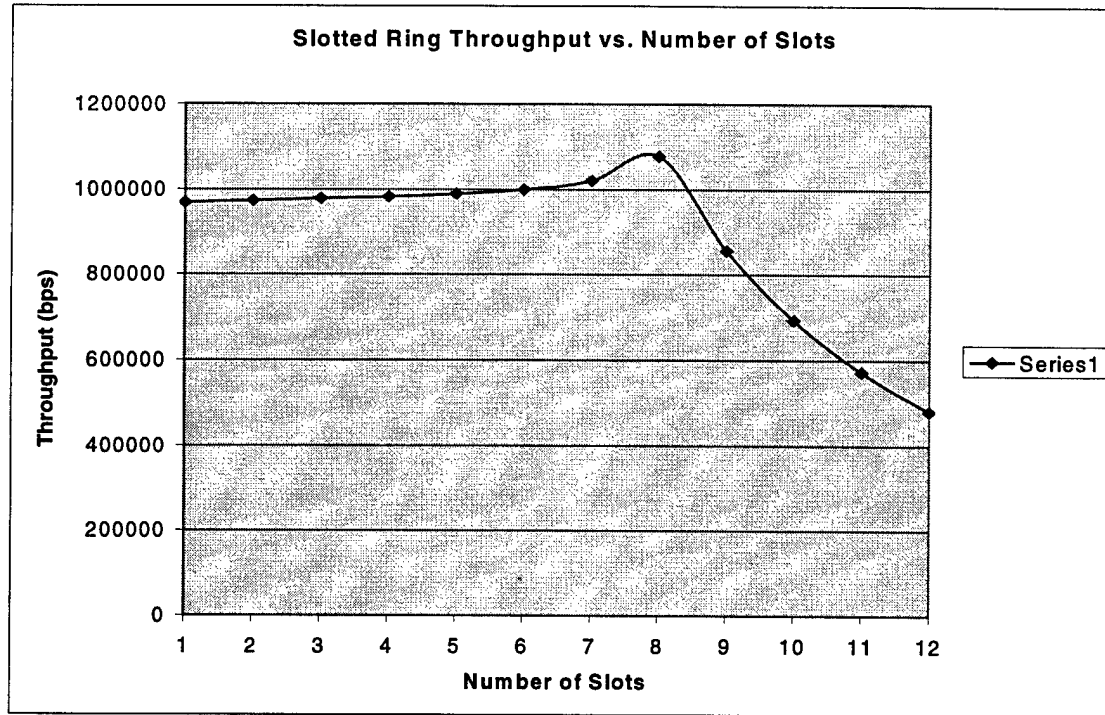


Figure 2.7 - Throughput of 8 Node Cluster versus Number of Slots. This test shows that a system with equal numbers of slots and nodes is most efficient. However, when operating below the saturation region, this result will be shifted to the left.

A convenient performance test was created to determine how well the slotted ring fares against the bidirectional array and the register-insertion ring. It fares very well since the test was optimal for a broadcast network. The test simply performed 100 barrier synchronizations in a row. The barrier synchronization uses a broadcast mechanism, which is the slotted ring's strongest quality. Figure 2.8 below shows the BONEs results for each of the simulations (The unabridged outputs are shown in the appendix. The final *Tnow* number and the name of the simulation are shown in bold. The slotted ring and the bidirectional array show similar results, but the bidirectional array performs better overall. The slotted ring is 7 percent less efficient than the bidirectional array and is somewhat more efficient than the register-insertion ring. The register-insertion ring is 11 percent less efficient as the bidirectional array. This test was composed to simulate the number of broadcast operations in a GDS architecture; therefore, one can expect performance in the neighborhood of these results.

Current Simulation Time is: 0.015549464151263 Slotted Ring System (8 Nodes)
Current Simulation Time is: 0.014521811157465 DPsA 8 Node Bi-Array
Current Simulation Time is: 0.016361624002457 DPsA RI Ring (8 Nodes)

Figure 2.8 – 100-Barriers Test. The 100-Barriers Test was created to compare the slotted-ring protocol to the bidirectional-array and register-insertion protocols.

2.4.3. Ratchet Network

One critical requirement of the final DPSA network is to provide global synchronization to all of the nodes. This requirement is based on the need to synchronize the analog-to-digital converters (ADCs). Without this synchronization, the beamform algorithm results would be incorrect. The physical limitations of the system prevent the use of a physically global clock. Thus, synchronization must be built into the communication protocol.

One method to create global synchronization is the use of a very deterministic protocol. An example is a version of the slotted-ring paradigm in which slots are passed around the ring at a predetermined size. The ratchet network is like a slotted-ring protocol with one exception: it is not limited by topology. The system may be set up in any number of ways including simple ring, bidirectional array, counter-rotating rings, etc. The network contains a slot for each node (or two slots for each node if nodes have two outputs, as do nodes in a bidirectional array or counter-rotating rings). The protocol involves synchronizing the slots so that they move uniformly from each node to the next (i.e., this protocol may be visualized as a ratchet, hence the name). This slot-per-node situation does limit the network in many ways, of which the most obvious is that it becomes a store-and-forward protocol. This limitation increases system latency in the network; however, latency may not be a problem if connections are buffered and if the system itself is latency-independent. Interestingly, the DPSA beamform algorithm itself is largely latency independent. However, the network must provide tight synchronization to ensure uniform sampling from the ADCs. Command requests and the use of hardware timers may provide these pseudo-low-latency messages. The ratchet network is ideal for this system because it may offer global synchronization or low-latency commands by the use of distributed timers, while at the same time providing latency-independent movement of data or control messages. The GDS system, in particular, may benefit from such a network protocol.

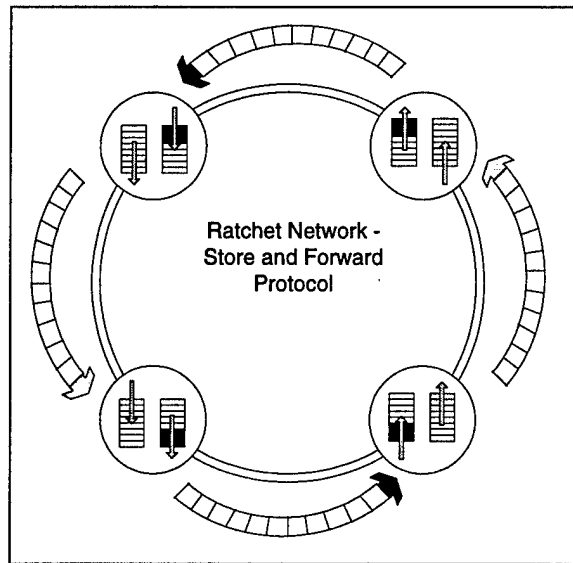


Figure 2.9 – Ratchet Network Protocol

The system operates in two phases, the first of which is the initialization phase. This phase synchronizes the network clock on both bit boundaries and word boundaries. Once word-boundary synchronization is successful, the nodes should all be operating synchronously in ratchet mode. The system enters this mode by allowing only the master to start sending initially by sending idle symbols. The idle size equals the entire frame of 60-120 bits, depending on

implementation. Once a downstream node receives some idles it begins to send idle symbols synchronized to the incoming idle frames. This process continues all the way around the ring or array and looks similar to filling an ice tray with water by filling the highest cube and letting the water run down to fill the next cube and then the next, etc. Since there will be propagation delays in the system, the first idles entering the master node in a ring system will not be synced with the master node's output. To compensate for this delay, the master node divides this delay by the number of nodes and sends a special control symbol (special idle) to all of the nodes to adjust the frame output time by the compensation time. After compensation, a propagated self-test symbol should be completely aligned and should arrive back at the testing node in sync with its own output symbols (ignoring the compensation factor, which is also added to the master node). A similar method may be used to synchronize a bidirectional array, but in this case, there are two master nodes of which one handles the compensation factor. The network "fills" one direction at a time (one unidirectional link, followed by the other).

Two types of messages may be passed around the network: data and command/control. Data must be passed in the data field and operate under the usual assumptions for a network protocol. Command and control, however, may be built into the Ratchet Packet Header. Control messages may be pseudo-synchronous with the aid of timers. The packet format is modeled after S-20 (CMT). The header packet is actually also a command packet and may use the 7-bit command window to issue low-latency commands. These commands can be part of the protocol itself for flow control or other network requests and may also be used in a proprietary manner for DPSA timed commands. These special DPSA operations will take advantage of hardware timers to issue low-latency requests such as "reinitialize nodes", "collect beamform result," etc. The data packets are also 20 bits in length and consist of 16 bits of raw data. If applying this packet format to a slotted contention scheme, the number of data packets in each slot is generally a fixed quantity between two and four and will be well known to all nodes in the finished DPSA. The bits of the proposed packet format are shown in Table 2.1 below.

<u>Ratchet Packet Header</u>		<u>Ratchet Packet Data Fields (for 4 field packet)</u>	
6	Command	16	Data
8	Destination	1	Always a One bit
1	Packet Status Bit (Occupied or Free)	2	Always a Zero bit
1	Parity Bit	1	Bias Bit (may be used to lower bias)
1	Always a One bit		
2	Always a Zero bit	16	Data
1	Bias Bit (may be used to lower bias)	1	Always a One bit
		2	Always a Zero bit
8	Scrub Counter	1	Bias Bit (may be used to lower bias)
2	Data Size		
4	Immediate Source Bits	16	Data
1	Service Bit *	1	Always a One bit
1	Parity Bit	2	Always a Zero bit
1	Always a One bit	1	Bias Bit (may be used to lower bias)
2	Always a Zero bit		
1	Bias Bit (may be used to lower bias)	16	Data
		1	Always a One bit
		2	Always a Zero bit
		1	Bias Bit (may be used to lower bias)
* unacknowledged or acknowledged message			
For four data fields Number of Total bits: 120 Number of Maximum Data Bits: 64 Maximum efficiency: 53%		For two data fields Number of total bits: 80 Number of Maximum Data bits: 32 Maximum efficiency: 40%	
		For one data field Number of total bits: 60 Number of Maximum Data bits: 16 Maximum efficiency: 26%	

Table 2.1 – Proposed Packet Format

The ratchet protocol improves synchronization at the cost of long latency. The slotted-ring model is ideal for a broadcast-heavy system, particularly GDS. Again, it can support two types of transactions for latency independent or synchronized signaling with generic data and command messages, respectively. Of course, high-level commands have the option to use either the specialized command packet or a data packet. The advantage of the former is the ability to simultaneously issue a command and pass a data message. Also the command messages can take advantage of hardware timers and processor signaling. Lastly, if a simple ring is unable to sustain the traffic requirements needed for this system, or if a ring is considered too hazardous for a fault-tolerant system, the topology can be adapted to a bidirectional array.

2.5. DPSA Architecture and Algorithm Performance Predictions

In the past year, the first experiments with beamformers over simulated DPSA network architectures were conducted. Using the ISE, the user can run simulations of different beamforming programs running on processors of varying speeds connected via networks of varying bit rates. To illustrate the variations that can be analyzed, two beamforming algorithms are compared. These algorithms are the medium-grain, non-pipelined FFT beamformer and the medium-grain, pipelined FFT beamformer. These algorithms are described in detail in the next chapter and in Appendix B.

The important details of these algorithms for the ISE simulations are in the communication stages of the algorithms. Both versions use the same basic algorithm. After each node computes the FFT on its data, a communication stage begins in which each node broadcasts its data column to all other nodes. At the end of the first communication stage, every node has the full data matrix comprised of a column from every node. Each of the nodes then computes the beamforming results for its share of the steering directions. At this point, the second communication stage begins each of the nodes sends its share of the results to a front-end node. The front-end node receives these results and finishes the iteration. The next iteration, which executes in exactly the same manner, immediately begins.

In the non-pipelined version, the communication stages occur exactly as described, all within the iteration. In the pipelined algorithm, the communication stages are pipelined in such a way as to overlap with computation. More specifically, after the Fourier transforms, each node sends its data column off to all the other nodes. However, the nodes do not block waiting for reception of this data. Instead, the nodes receive the data sent the previous iteration. This data should have already propagated through the network and been buffered by the communication implementation without the knowledge of the beamformer program. The data just sent for the current iteration will not be received in the current iteration. In this manner, the nodes avoid any blocking and begin their steering direction calculations sooner than if they had needed to wait for the data. Thus, the steering direction calculations for the data from the previous iteration are overlapping the communication of the data for the current iteration. The same situation occurs in the second communication stage, where the results from the current iteration are sent to the future iteration and the front-end node receives the previous iteration's results without the need for blocking.

Simulation results are provided for 8-node bidirectional array networks. The link speeds between the nodes are varied between 2.5 Mbps and 10 Mbps. The performance of the processors on each node is varied between the performance of a one-quarter-speed UltraSPARC-1/170 and the performance of a full-speed UltraSPARC-1/170. The beamforming algorithm is set for 1 iteration, 64 samples per iteration, and 91 steering directions per iteration. In the case of the pipelined program, 3 iterations are run so that the pipeline can be filled and drained. In this situation, the sole purpose of the first iteration is to send the FFT samples to the timed iteration, and the sole purpose of the last iteration is to sink the partial results sent out of the timed iteration.

Only the one full iteration in the middle is timed. The results for the non-pipelined program are shown in Figure 2.10. Figure 2.11 shows the equivalent results for the pipelined version.

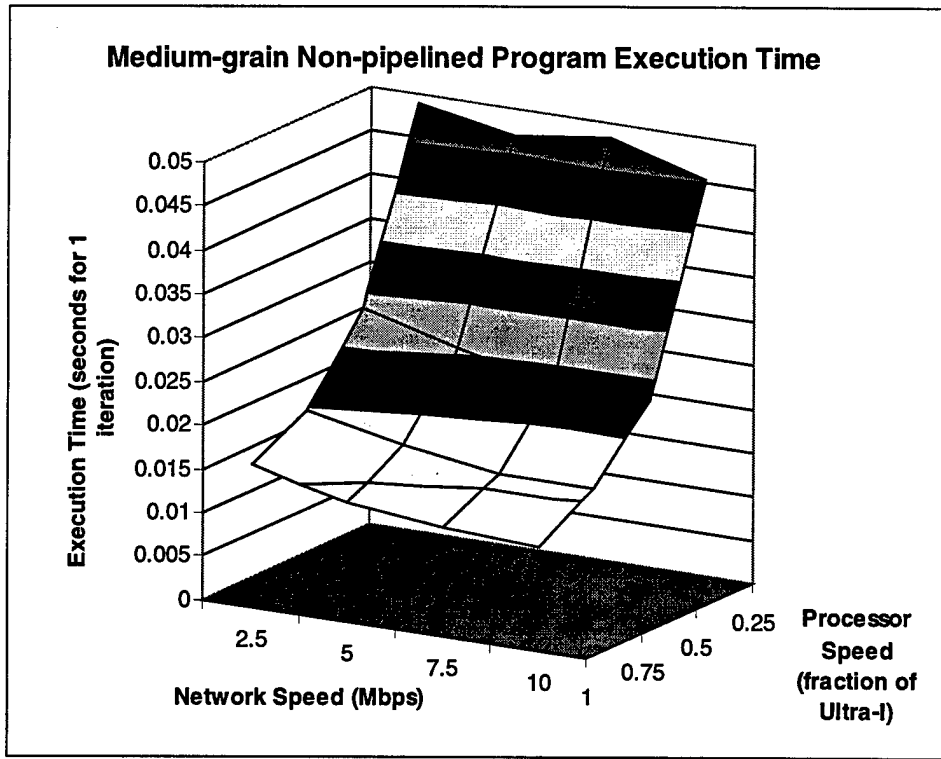


Figure 2.10 – Execution Time for Non-pipelined Program over Different Network and Processor Speeds

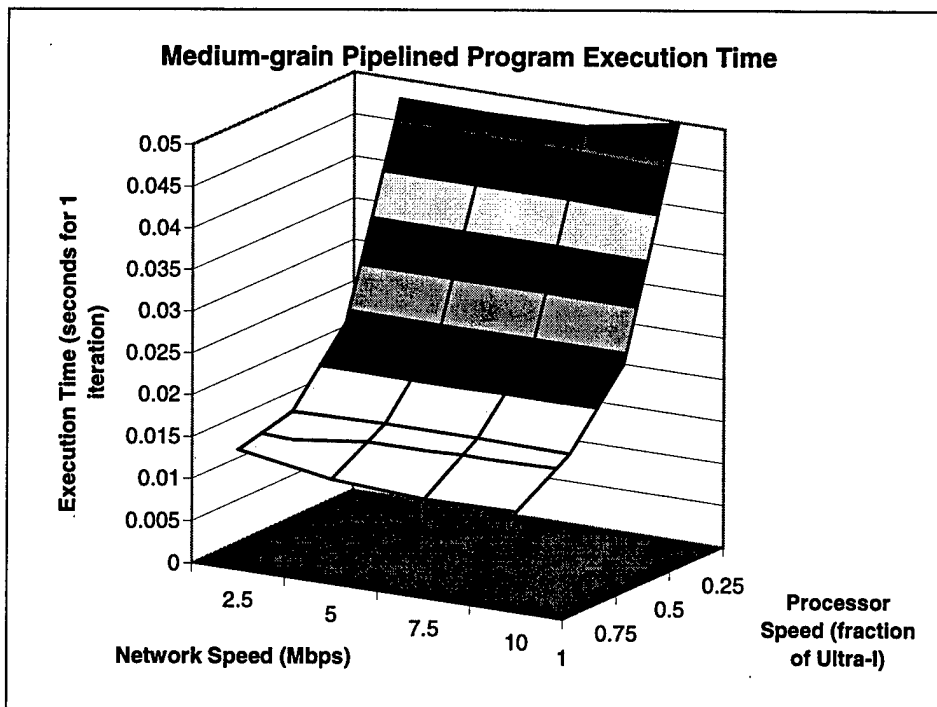


Figure 2.11 – Execution Time for Pipelined Program over Different Network and Processor Speeds

In the non-pipelined program, when the network speed is set to 10 Mbps, the execution time for 1 iteration of the program ranges from 10 ms for the fastest processors to 45 ms for the slowest processors. When the network speed is slowed to 2.5 Mbps, the execution time with slow processors increases to 50 ms. In addition, for the fast processors, the slow network causes the execution time to slow to just below 15 ms. The ISE results show that slowing the network speed on an array running the non-pipelined program will decrease the performance a small, though measurable, amount. The slowdown due to a slower network is mostly independent of the processor speed, indicating the independence of communication and computation in the non-pipelined program. The ISE also shows that the execution time is very dependent on the processor speed no matter what the network speed. When using a fast network, decreasing the processor speed for the non-pipelined program by 75% decreases the performance by about 75%. This result indicates that for a network speed of 10 Mbps, the execution time is dominated by the computational requirements. When a network speed of 2.5 Mbps is used, the decrease in performance is only 67 percent, illustrating that the communication time has become an important factor and that the execution time is less dominated by computation.

The results for the pipelined program also show interesting trends. The most important result is that the execution time is mostly independent of the network speed for the values simulated. In most cases, for a given processor speed, the network speed does not have a significant impact on the algorithm performance. This phenomenon is, in fact, the goal of the pipelined program. Rather than requiring processes to block to wait for the data propagation through the network, the communication is overlapped with computation on the data from the previous iteration. The exception occurs when the network is very slow and the processors are very fast. In this situation, the time required to complete the computation for the previous iteration data is not long enough to allow the complete propagation of the current iteration data through the network, and the processes must block despite the pipelining.

For fast networks, the real-world difference in execution time between the non-pipelined version and the pipelined version is small. This result again illustrates the fact that for fast networks, the computational time dominates the communication time; the pipelining has little effect. For slow networks, the difference between the two algorithms is more noticeable. For a network speed of 2.5 Mbps and processor speed of 50 percent of an UltraSPARC, the execution time for the pipelined version is 16 percent faster than that of the non-pipelined version. For 75-percent-speed UltraSPARC processors, this fraction improves to 19 percent. As processor speed continues to increase, the relative performance of the pipelined program over the non-pipelined program will continue to increase. Only when the processor speed becomes so fast that the overlap with communication is not complete does the performance factor of the pipelined version over the non-pipelined version stop improving.

With these results in hand, the performance differences between the two programs can be compared against the real-world requirements of the sonar array. If slow networks must be used for the sonar array, the pipelining of the second program may provide enough performance improvement to warrant the additional overhead needed to buffer messages when nodes are busy computing. On the other hand, if a fast network can be used, then the non-pipelined version performs as fast as the pipelined program. In this situation, the additional complexity required in the pipelined version would make it a poor choice for the sonar array.

These simulations will be extended for 16 nodes, 32 nodes, or more. The data gathered in such experiments will provide more detailed information on which to make conclusions about the best algorithm for the sonar array. In addition to running larger simulations, beamforming algorithms other than the two medium-grain versions presented here and networks other than the bidirectional array will be simulated. These simulations will allow more quantitative results to be seen, and 8-node results will not have to be extrapolated for the large DPSA.

2.6. Conclusions

The ISE promises to provide a unique and interesting rapid virtual prototyping tool and has delivered in several initial scenarios. The simulation of any user MPI program over any modeled network architecture is a powerful method for design, verification, and validation. With the software simulation tools developed here, new or existing MPI programs using standard sends and receives may be tested and timed over a desired network architecture without the need to create actual prototype hardware.

The ISE strives to be user-friendly. The MPI application programs do not usually need any modifications to run over the BONEs network. The ISE includes several ease-of-use features, such as the fast-forwarding function, which speeds up the simulation when there are no outstanding requests on the network (explained in Appendix A). The ISE aims to make the process of virtual prototyping as convenient to the developer as possible.

With the detailed information provided by the ISE for the conventional beamforming programs, the performance of the final autonomous sonar array can be simulated with a high degree of fidelity. In addition to the predictions made for the DPSA project, the ISE shows excellent promise to be a cutting-edge development tool for future work in developing more complicated distributed sonar applications with advanced beamforming methods on all forms of arrays.

3. Parallel Conventional Beamforming

The beamforming algorithms chosen to parallelize initially for the DPSA are based on a standard radix-2 Fast Fourier Transform (FFT) beamformer that will be reviewed with references for the reader. Parallel algorithms with different levels of granularity were developed via coarse-grained iteration decomposition and medium-grained steering decomposition. In addition to a network-independent implementation (in which no assumptions are made about the interconnection topology) all algorithms were mapped to unidirectional-array, bidirectional-array, and ring network implementations. These algorithms are described textually and graphically prior to analyzing models of computation and communication. These models will provide quantitative support for intuition on how each of the algorithms should perform. Finally, the results of actual parallel code are presented and discussed. The beamform algorithm timings were collected from a cluster network of eight dual-CPU SPARCstation-20/85 workstations connected via 155-Mbps-per-link (OC-3) ATM, a basic production and research network configuration.

3.1. Sequential Conventional Beamforming

The essential operation of beamforming is to sum the manipulated outputs from many spatially separated sensors. The spatial separation can be accounted for in the time domain using time delays or in the frequency domain using phase shifts; both methods produce an equivalent output. It can be shown that the computational complexities of the methods are equal, and it is therefore redundant to the purpose of this document to consider both methods. Working in the frequency domain is commonly known as an FFT beamformer and is the method analyzed in this document. No attempts have been made to optimize the baseline sequential FFT beamform algorithm other than through parallelization. Figure 3.1 shows the flowchart for the sequential algorithm. The FFT code was adapted from [MORG94].

The first computation in each sample set is to calculate the window functions for each node and to multiply that node's data samples by that factor. Next, the algorithm performs a FFT on the windowed data from one node at a time. The algorithm then enters a loop that executes once for each steering direction. The steering directions in this project all range from -90 to +90 degrees from perpendicular to broadside, with adjustable search angle increments. For each steering direction of the loop, the algorithm multiplies the transformed data from each node by a node-dependent steering factor, which is a function of the node's location and the current steering direction. It is important to note that this steering factor multiplication is essentially the same operation as correlation by plane-wave replica vectors. It is assumed that the steering factors are recomputed for each iteration. The next step is to sum the corresponding samples across all nodes, sample by sample, for all samples in the sample set. The algorithm then inverse-transforms this summed data matrix. A single-valued result for the current steering direction is obtained by finding the magnitude of the resulting signal. The algorithm stores this value before looping to the next steering direction. At the end of the algorithm, all the values obtained from the steering directions can be plotted versus steering direction so that the signal power can be seen spatially by the user. The entire process is repeated for successive iterations with new sample sets.

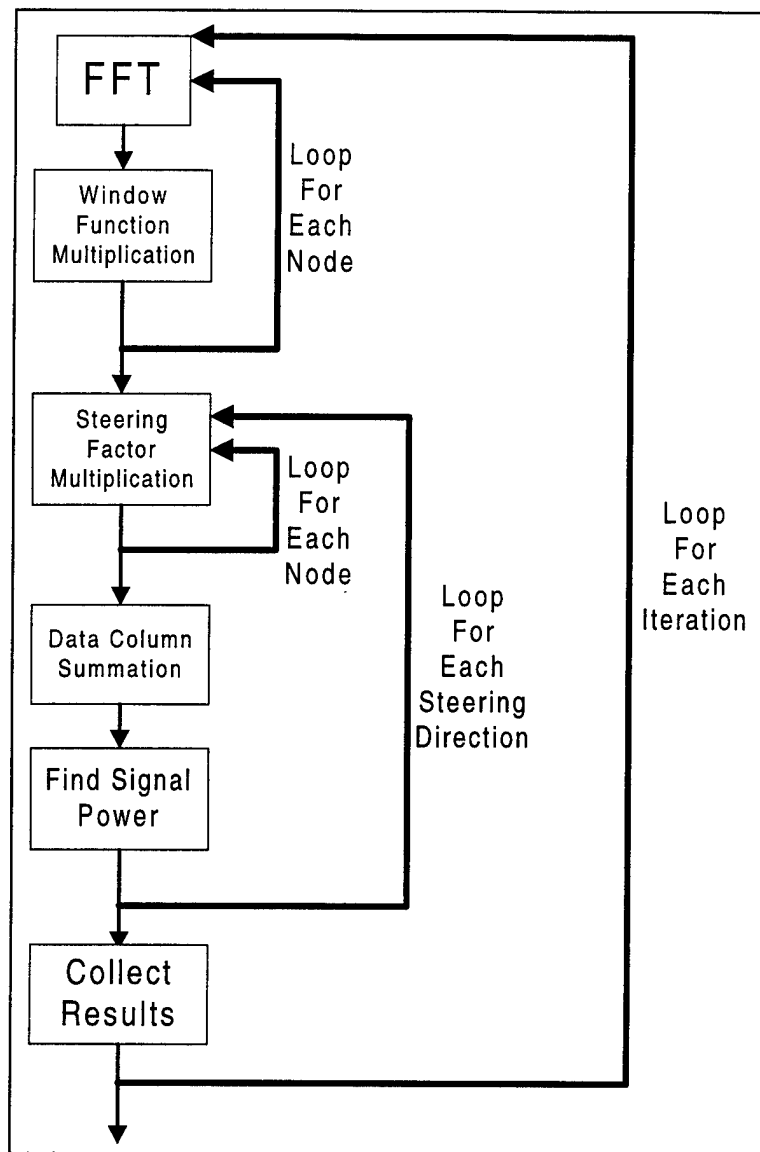


Figure 3.1 - Flowchart for Sequential FFT Beamformer. After each node collects acoustical sonar data, a single processor performs the sequential algorithm. The final step of collecting results is done by plotting the relative magnitude of the signal power at each steering direction for a single iteration.

3.2. Parallel Conventional Beamforming Algorithms

Coding of the parallel algorithms was divided into two methods related to network topology: network-independently in which no assumptions are made about the underlying interconnection topology and with network-dependencies. The network-dependent coding was termed a *rudimentary network simulation*. The rudimentary network code simulates the complexity of communication over different network architectures by making communication calls over no more than one link at a time. Nodes cannot send data directly to any arbitrary node in the array, but must follow the communication path that would exist in the physical array. For example, forcing each node to send only to the node immediately downstream simulates the implementation for a register-insertion ring architecture. In the network-independent programs, it is possible for any node to send directly to another node with one communication call by way of a

completely-connected point-to-point network, as opposed to the more restrictive communication patterns used in the rudimentary network code. The motivation for coding a given algorithm with these different methods is that the algorithm can be run over more than one architecture to determine the feasibility of using that algorithm on different architectures. Though the simulation is not exact in the timing of particular real architectures, this method is useful for general comparisons between different implementations of the same algorithm.

It is important to maintain constant simulation overhead across multiple algorithms to assure valid comparisons. These comparisons are possible because all network-dependent programs implemented here, including the baseline, were coded with the same rudimentary network structure. The network-dependent parallel algorithms can be compared with the network-dependent baseline strictly on the basis of the quality of the algorithms. The same is done for the network-independent programs. The use of high-performance workstations instead of beamforming processors provides useful data in gauging algorithm performance since the algorithms are being compared only against each other and are using the same overhead structure and assumptions.

The acronyms used for the different algorithms may become confusing. The reader is referred to Table 3.1 below for a summary of the algorithm properties. A description of each of the algorithms is then outlined.

Algorithm	Granularity	Communication	Network	Parallelism
CPUNF	Coarse	Unidirectional	Independent	[FFT, window multiplication]
CPUF	Coarse	Unidirectional	Unidirectional Array	[**]
CPNF	Coarse	Full Capability	Independent	[**], pipelined front-end node
CPBF	Coarse	Full Capability	Bidirectional Array	[**], pipelined front-end node
CPRF	Coarse	Full Capability	Unidirectional Ring	[**], pipelined front-end node
MPUNF	Medium	Unidirectional	Independent	[**], steering directions
MPUF	Medium	Unidirectional	Unidirectional Array	[**], steering directions
MPNF	Medium	Full Capability	Independent	[**], steering directions, signal power
MPBF	Medium	Full Capability	Bidirectional Array	[**], steering directions, signal power
MPRF	Medium	Full Capability	Unidirectional Ring	[**], steering directions, signal power

Table 3.1 - Summary of algorithm acronyms and properties.

3.2.1. Coarse-grained Parallel Algorithms

Two types of algorithms were developed which are coarse-grained in nature. The first type is designed solely for a unidirectional linear array in which communication can only proceed from the most upstream node to the most downstream node. The two algorithms of this first type are the coarse-grain unidirectional FFT beamformer (CPUF) and the coarse-grain unidirectional network-independent FFT beamformer (CPUNF), which are identical except that the former contains a rudimentary network structure where messages must be forwarded through adjacent communication paths. The latter uses single MPI calls instead of node hopping to send messages downstream. The second type of algorithm is targeted toward fully-connected bounded-degree networks (FCBDNs): a ring, a bidirectional linear array, and a network-independent version called the coarse-grain full-capability FFT beamformer with ring network (CPRF), the same with

bidirectional network (CPBF), and again the same with no rudimentary network (CPNF), respectively. Again, the network-independent version is identical to the versions with rudimentary ring and bidirectional networks but places no restrictions on and makes no assumptions about communication paths.

3.2.1.1. Coarse-grained Unidirectional Algorithms

Since communication is only one-way, the only node capable of receiving data from all nodes is the most downstream node. The most downstream node in the array then must always do the most work because the algorithm requires a summation of all nodes' data. This fact severely limits the degree of parallelism possible.

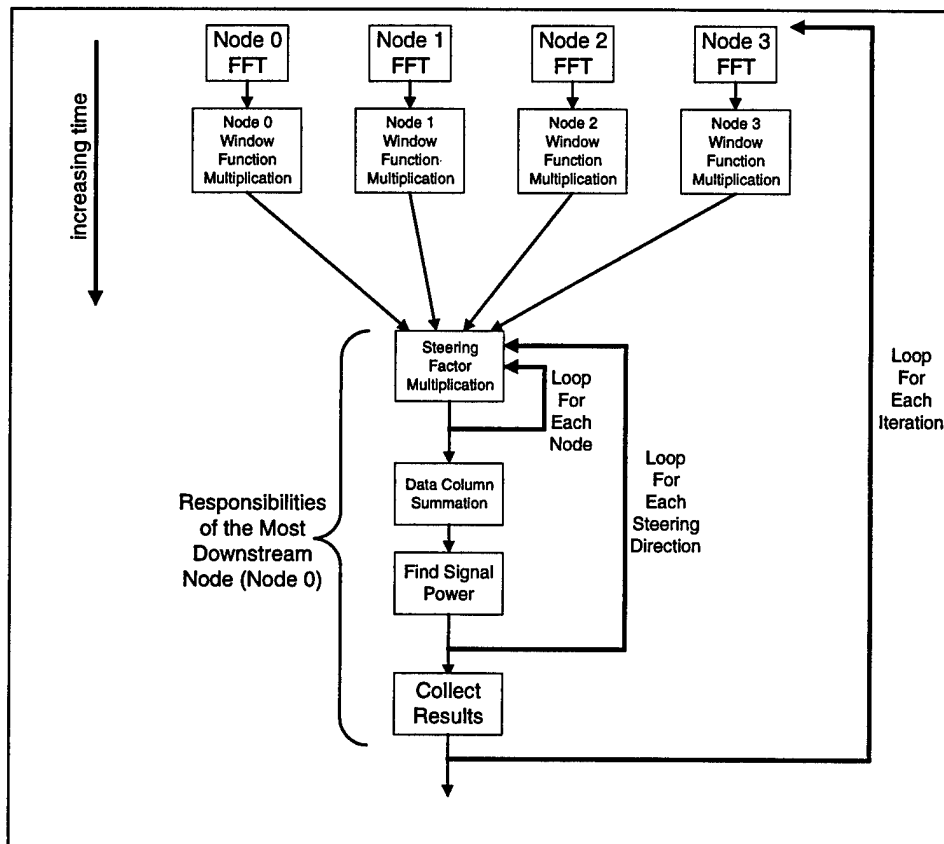


Figure 3.2 - Flowchart for CPUNF, Coarse-grained Unidirectional Network Independent FFT Beamformer. The front-end node after each node has performed window function multiplication carries out the main portion of the algorithm, which must be the most downstream node (Node 0). Notice the only difference between this algorithm and the sequential one shown in 4.1 is data parallelism during the FFT and window multiplication stages.

The parallelism technique exploited for the coarse-grained unidirectional algorithm is data parallelism where same operation is performed on different data at the same time. The initial FFT and window-function multiplication operations are done simultaneously in a data-parallel manner. This stage of operation shall be termed the *opening sequence* and is found in all of the algorithms. Once each node has completed its opening sequence, the nodes send their data down the linear array to the first node. The first node then carries out the remainder of the sequential algorithm, which means it executes the loop for all the different steering directions. Figure 3.2

shows the flowchart for how the parallelism in this algorithm takes place in the version with no rudimentary network simulation, where blocks lined up horizontally are computed simultaneously.

One important aspect of this algorithm is the communication pattern; the amount of communication increases linearly with the number of array nodes. This result is due to the fact that the front node can receive from only one node at a time, and all receives must be made one after another. The nodes cannot reduce communication time by combining passing data because each set must be multiplied by node-dependent factors for each steering direction.

For the version of this algorithm with the rudimentary network simulation, the communication pattern is modified from that shown in Figure 3.2. Instead of sending all data columns directly to Node 0 with a single MPI call, the data is communicated via a growing "freight" train. The most upstream node sends its data column to its downstream neighbor. This node then appends the received data column to its own and sends the two-column matrix to the next downstream node. The process continues until the most downstream node receives the entire matrix.

3.2.1.2. Coarse-grained FCBDN Algorithms

The next set of algorithms was developed for a fully-connected bounded-degree network (FCBDN), which means that any node can communicate with any other node by way of some communication path through the nodes. The algorithm no longer needs to be concerned about which nodes can communicate with which other nodes. This algorithm takes advantage of networks such as rings and bidirectional linear arrays. As opposed to the unidirectional case where the first node always had to do more work than the other nodes, the algorithm for the full-capability network allows the node doing the front-end work to be floating. That is, rather than forcing all data to be sent to the first node every time, the data can be sent to a different node each iteration. The concept is that of a floating front-end processor and all nodes use a deterministic algorithm to figure out which array node is acting as the front-end for a particular iteration. This ability makes possible better link utilization and better processor computational usage. The back-end node no longer has the least work to do because eventually it will be designated the front-end to which all other nodes send their data to be manipulated.

In addition to data parallelism, these communication enhancements introduce a type of agenda parallelism, termed *specified agenda parallelism*. In agenda parallel applications, the working processes figuratively reach into a grab bag of tasks that need completion. When they pull out their assignments, they begin to work on them. In the *specified agenda parallel* concept, the tasks that the working processes pull out of the grab bag are not random choices. Instead, a selection algorithm is created so that the tasks pulled out of the bag are in a specified order.

The general FCBDN algorithm modifies the algorithm from the coarse-grained unidirectional program and implements the floating front-end. The nodes all take their own transforms and compute their own windowing function factor to multiply into their data. Then each node determines who is the front-end by using a specified formula. This formula simply follows a "round-robin" scheme in which the front-end for the next iteration is the neighboring node to the front-end for the current iteration. Once a front-end is chosen for a particular iteration, communication then proceeds to that front-end just as was done in the unidirectional version. The front-end node then starts the loop for the several steering directions and gathers the results.

The algorithm follows the tradition of pipelining, where one operation does not need to be completed before the next operation is started. Pipelining applies to this algorithm in that the nodes can collect another sample set from the acoustic transceivers and begin the second iteration

before the first iteration is completed. At the beginning of the second iteration, the node doing the first iteration front-end work stops what it is doing for just enough time to FFT the second iteration data, multiply by the windowing factor, and send it off to the second-iteration front-end. Once the data has been sent to the new front-end, the node resumes front-end work for the previous iteration data. The process continues for consecutive iterations, where the front-ends from the previous iterations stop their work for a moment. As the iterations progress, more and more nodes are doing front-end work for their respective iteration numbers, all in different stages of completion. The flowchart is shown in Figure 3.3.

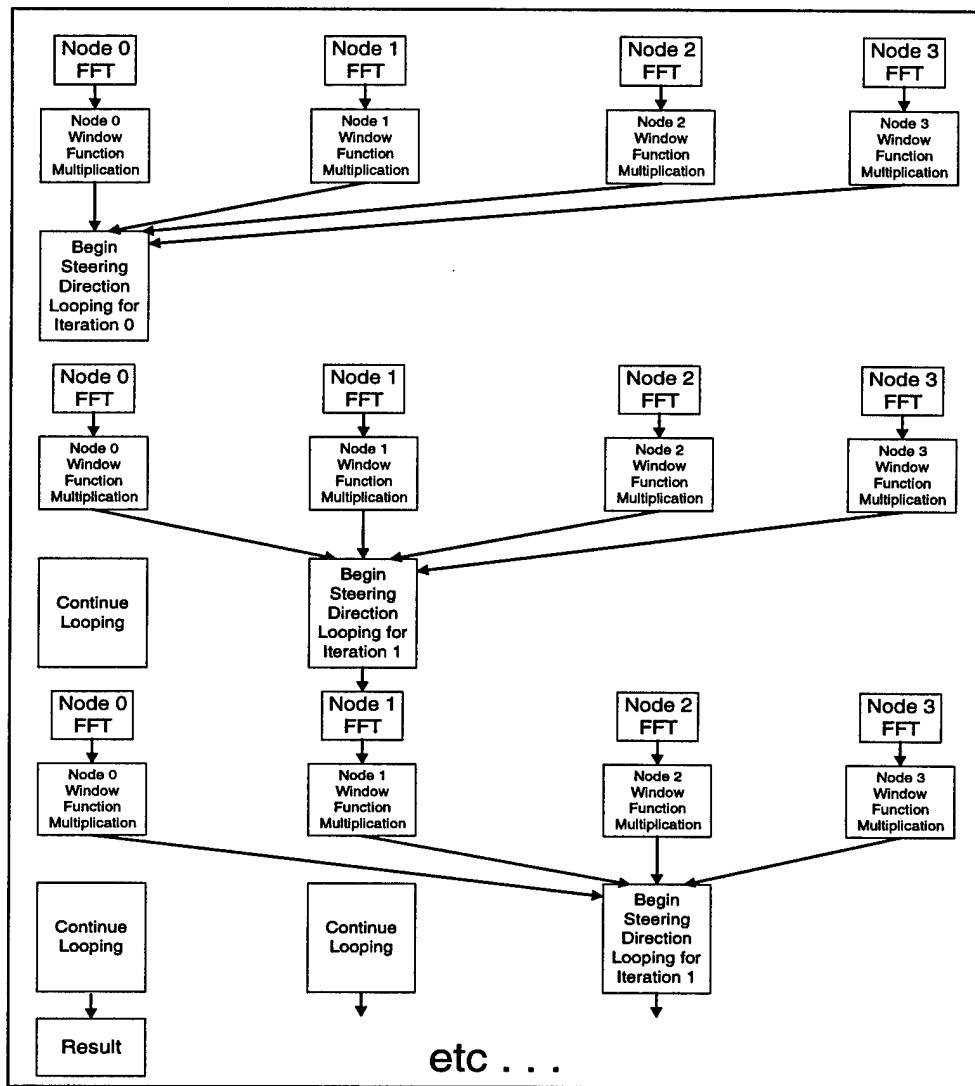


Figure 3.3 - Flowchart for the CPNF, Coarse-grained Network Independent FFT Beamformer. Although it looks more complex than the flowchart for coarse-grain unidirectional program shown in Figure 3.2, the only change made is that there is no longer a downstream node that must be the front-end node. The sequential tasks performed previously are now done in a pipeline fashion by all of the nodes.

In order to avoid assigning new front-end work to a node still doing front-end work from a previous assignment, the algorithm must assure that there is no wrapping of the front-end assignment until such time as a complete iteration of work can be finished. This algorithm is implemented by having the front-end processors stop calculation on front-end work after a

specified number of steering direction loops to start the next iteration. For example, if there are 8 nodes scheduled to be front-end nodes and 91 steering directions to check for each iteration, then the front-end nodes will stop their front-end work after every 12 executions of the steering direction loop. Therefore, by the time the assignment of front-end nodes has wrapped around to a node already assigned, that node will have finished the 91 steering directions (since 12×8 equals 96) and be free to take on responsibilities of another assignment.

In the network-independent version of this algorithm, all nodes use a single MPI call to send their data column to the current front-end node. In the ring version, the communication proceeds in the growing freight train from the node just downstream of the front-end node, around the ring, to the front-end node. In the bidirectional version, the unidirectional communication is started at both ends, so that the current front-end node receives part of the complete matrix from the right and part from the left.

3.2.2. Medium-grained Parallel Algorithms

Two types of algorithms were also developed which are medium-grained in nature. The types are completely analogous to the coarse-grained versions and differ only in the method and granularity of parallelism. The medium-grain unidirectional programs (with unidirectional rudimentary network and with no network) belong to the first type, and the medium-grain full-capability programs (with ring rudimentary network, bidirectional rudimentary network, and no network) belong to the second type.

3.2.2.1. Medium-grained Unidirectional Algorithm

The medium-grain unidirectional algorithm decomposes the steering direction loop previously performed by a single node and makes the operations in the loop part of the responsibilities of the several nodes. The major goal that this reorganization accomplishes is to complete the summation as the communication is progressing in order to improve the degree of parallelism and spread the computational workload. Rather than sending column after column to the front node, each node receives a column from upstream for a particular steering direction, adds its column for that direction to it, and sends the single summed column downstream. A granularity knob was built in to allow multiple steering direction columns to be communicated at the same time. Rather than sending just a single column to be summed, each node works on several steering directions. With more steering directions communicated each time, the loop occurs fewer times. There is, however, a drawback in that the medium-grained form of communication (with summation) must be carried out multiple times for each iteration. There is a trade-off between communicating with the coarse-grained algorithm but only doing it once each iteration (each sample set), and communicating considerably fewer values in the medium-grained algorithm but doing it several times per iteration. The flowchart for the rudimentary network algorithm is illustrated in Figure 3.4. For the network-independent version, all nodes send their steering direction columns to Node 0, which receives them one at a time and sums them with the columns already collected.

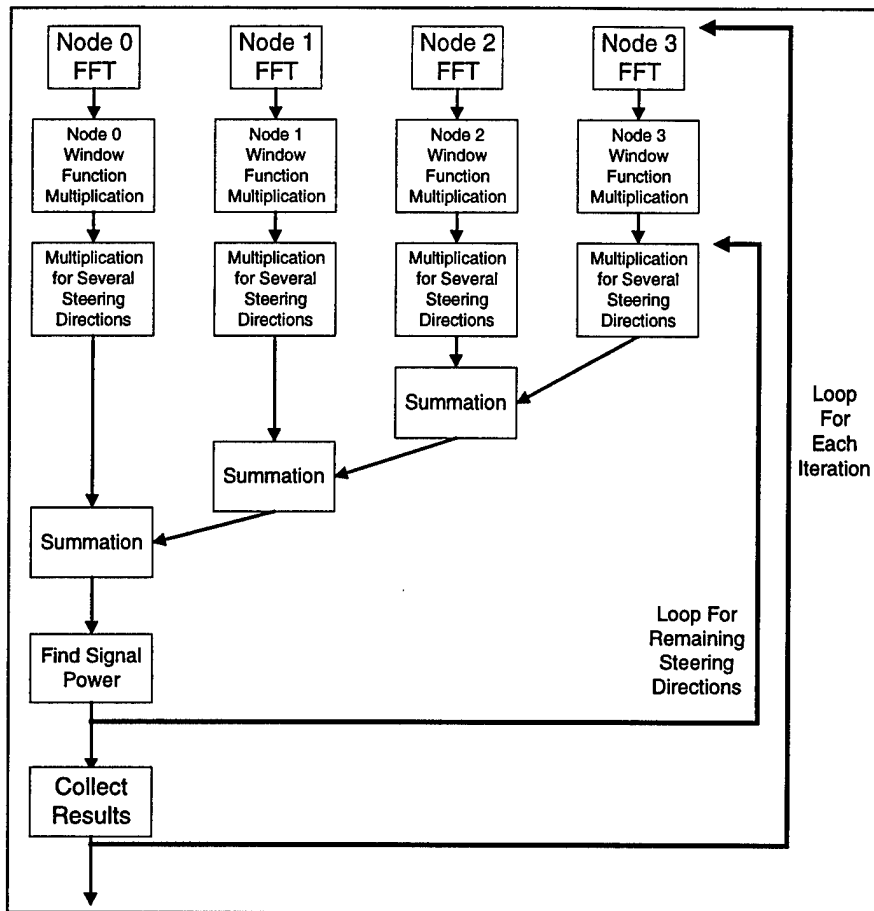


Figure 3.4 - Flowchart for the MPUF, Medium-grain Unidirectional FFT Beamformer. Parallelism is applied at the level of the steering directions. Each node is responsible for multiplying the data by a predetermined number of steering direction factors, relieving the front-end node from doing this sequentially. Although less computation is done by the front-end processor, more communication stages per iteration must occur.

3.2.2.2. Medium-grained FCBDN Algorithm

The medium-grained algorithm developed for fully-connected bounded-degree networks exploits steering direction decomposition. It incorporates both data parallelism and result parallelism. The data parallelism arises from the fact that the processes work with local data independently whenever possible. The algorithm also employs result parallelism in that the processes each calculate separate parts of the result which are collected at the end of the iteration.

First, data parallelism occurs in the initial transform and window factor multiplication operations in each iteration, just as in the previously described algorithms. Once each node has finished these steps, the algorithm prepares to split up the steering direction loop.

The result parallelism arises from the fact that each node calculates the beamform result for a certain number of the total set of desired steering directions. However, before this process can proceed, any participating node must have a copy of the data from all nodes to make a correct calculation. As such, this algorithm incurs considerably more communication than the coarse-grained algorithms and the interconnection scheme will have a more significant effect on performance.

For a ring network, the algorithm specifies that a growing freight train is used for the communication around the ring. The freight train communication begins when the back-end node sends its data column (which has already been transformed into the frequency-domain and multiplied by the weight factor) to the second-to-last node. The second-to-last node then appends this column to its own, resulting in a matrix with 2 columns. This node then sends the matrix downstream, where the next node again appends its data column. The process continues until the front node has received the freight train from all upstream nodes. At this point, the front node has all the data and can proceed with calculating the results for its share of the steering directions; however, the other nodes cannot begin until they receive the entire freight train. Therefore, the front node uses the link between it and the rear node to start the freight train around the ring once more. In this case, however, the train's length is decreasing as it passes between nodes. After the front node receives the full matrix, it does not need to send the entire matrix to the rear node because the rear node already has its own column stored. Likewise, the rear node does not need to send the entire matrix to the second-to-last node because the second-to-last node already has its own data column and that of the rear node. At each node in the second half of the communication, the freight train could decrease in size by one column. The flowchart for the ring algorithm is shown in Figure 3.5 and the communication strategy for the program is shown in Figure 3.6.

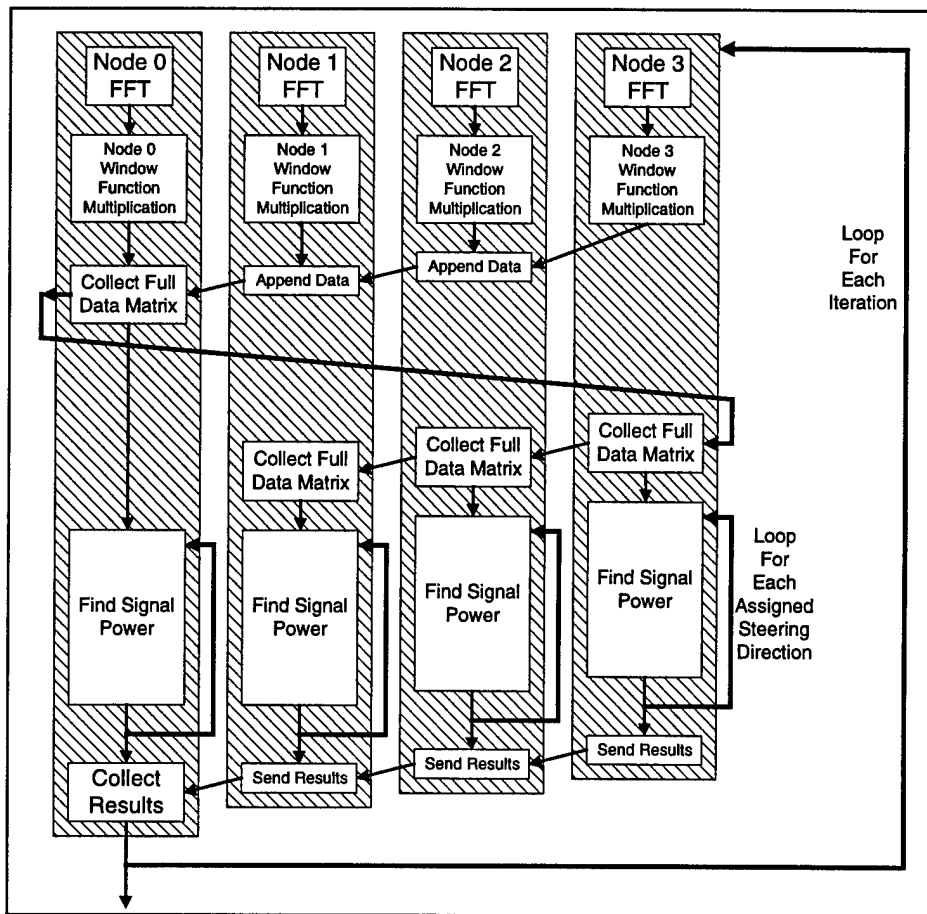


Figure 3.5 - Flowchart for MPRF, Medium-grain Full-capability FFT Beamformer with Rudimentary Ring Simulation. All nodes can receive data from any other node so that the ability of parallelizing the steering factor multiplication is extended to also being able to calculate the signal power for the given steering directions.

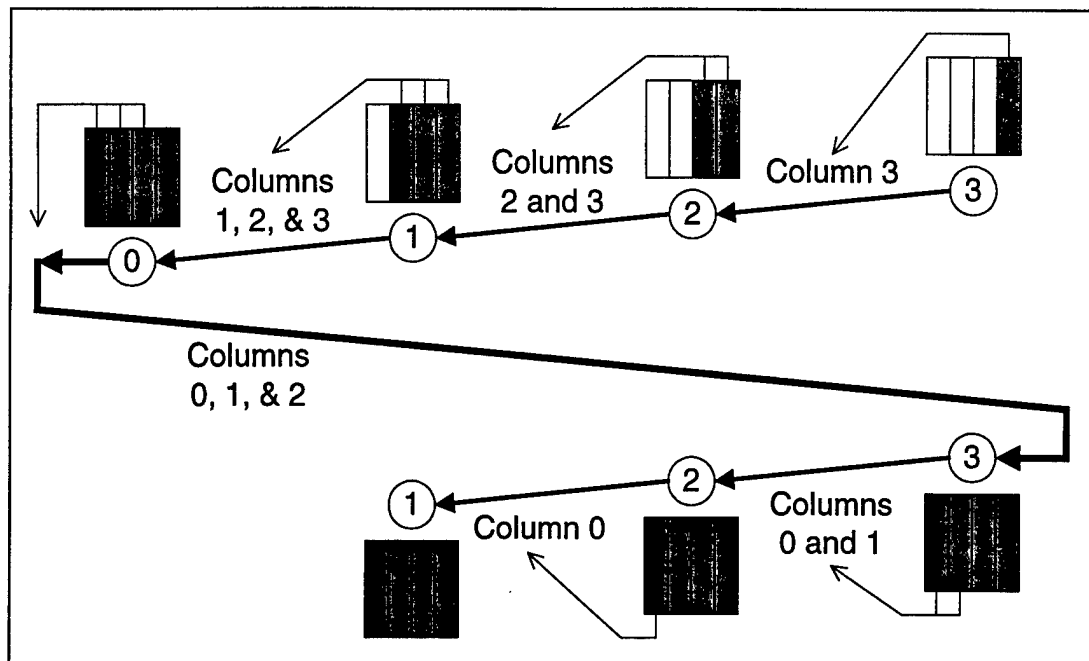


Figure 3.6 - Ring Communication for MPRF, Medium-grain Full-capability FFT Beamformer. The communication pattern shows that the data must travel in a growing freight train to the front-end node and in a shrinking freight train around the ring until all nodes have the full data matrix.

The algorithm and communication pattern for the bidirectional array are shown in Figure 3.7 and Figure 3.8, respectively. A growing freight train, as discussed above, is started at both ends of the array. Once a node receives both trains, it has the full data matrix. For the network-independent version, each node broadcasts its data column to all other nodes, then receiving a column from each other node, accomplishes the communication. At the end of the process, each node will have the full data matrix.

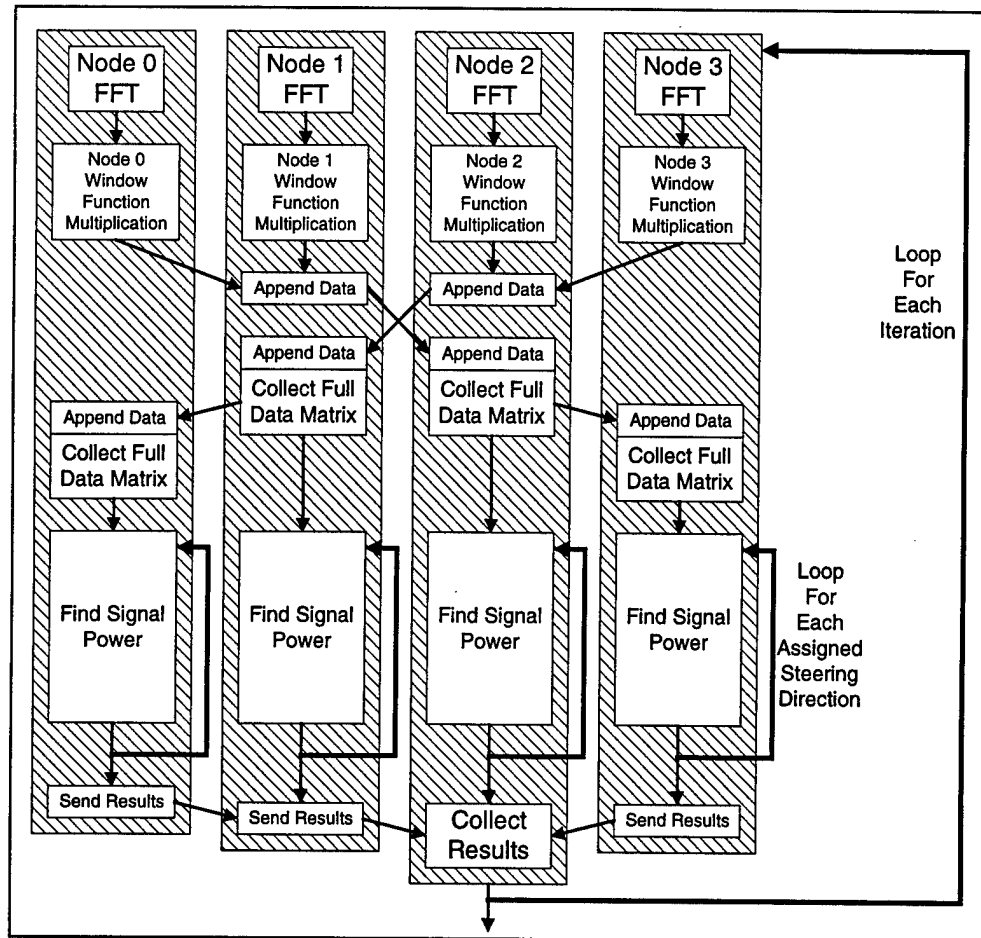


Figure 3.7 - Flowchart for MPBF, Medium-grain Full-capability FFT Beamformer with Rudimentary Bidirectional Array Simulation. This algorithm differs from the others in its class by the communication pattern of the data prior to finding the signal power. This pattern is shown in more detail in Figure 3.8.

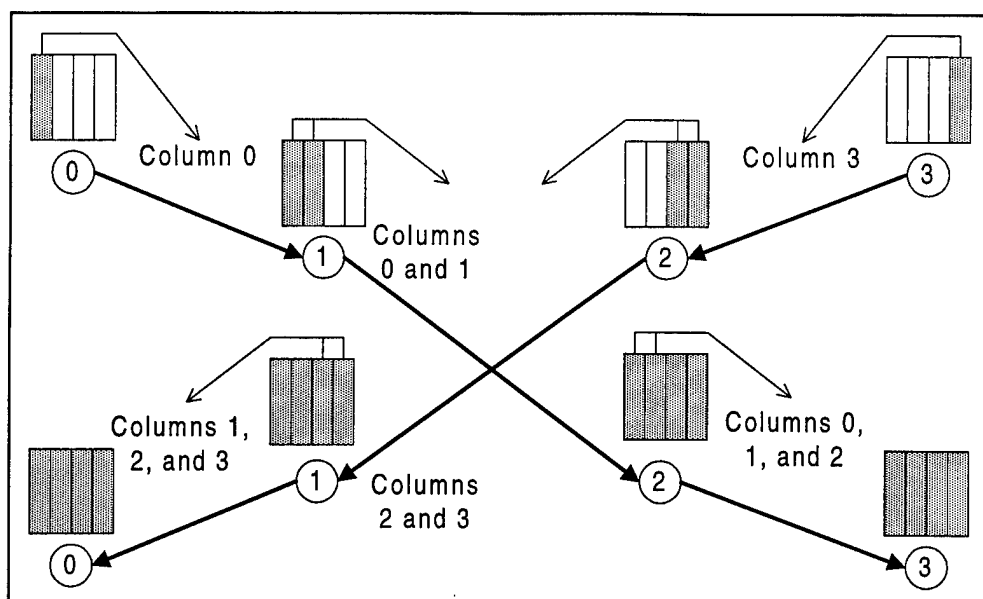


Figure 3.8 - Bidirectional Array Communication for MPBF, Medium-grain Full-capability FFT Beamformer. Because the topology is a bidirectional linear array, communication is overlapped by sending data in both directions simultaneously.

Once the communication is complete, the result-parallel aspect of the medium-grained algorithm is employed. Each node begins work on its share of the steering directions. The steering direction on which a node is to begin computing is determined by that node's relative location from the front node. The number of steering directions a node is to compute is based on dividing the total number of desired steering directions by the number of nodes. When the number of nodes cannot evenly divide the number of steering directions, the nodes closer to the front calculate more directions. For example, if 4 nodes are to calculate 6 steering directions, node 0 calculates directions 0 and 1, node 1 calculates 2 and 3, node 2 does 4, and node 3 does 5.

After a node is finished computing the results for its steering directions, it must communicate them to the front node for final collection. For the ring and bidirectional array, the rear node initiates the communication by sending its share of the results in a sub-array to the second-to-last node. When that node is done, it will receive the sub-array from upstream, append its result sub-array to it, and send the collection downstream. The procedure continues much like the growing freight train of the data matrix communication, with the exception that once the front node receives all the results, the iteration is over. For the network-independent version, each node simply sends its result sub-array with one MPI call to the front node.

3.3. Performance Prediction

Analytical models were built for many of the decomposition methods described for the DPSSA beamform algorithms. These models provided foreknowledge of any algorithm's complexity and likely speedup. In addition, the implemented algorithms were also compared to justify their performance. Of course, the fidelity of the models is low, so actual performance times cannot be compared, but the relation between one analytical model and another should loosely hold from one program to another.

The performance of the parallel beamformers is dependent on a moderate number of free, mutually-independent parameters. These parameters were combined into predictive models of algorithm performance. Instead of reducing the predictive models to order notation (i.e. $O(N)$),

the models retain the few free parameters in order to obtain better fidelity. A model was built as a combination of computation and communication predictors allowing some degree of overlap. The general case is described by Equation 3.1.

$$\text{Time} = \min\left(\frac{\text{Computation}}{\text{FLOPS}}, \frac{\text{Communication}}{\text{Bandwidth}}\right) \cdot (1 - \rho) + \max\left(\frac{\text{Computation}}{\text{FLOPS}}, \frac{\text{Communication}}{\text{Bandwidth}}\right) \quad \text{Equation 3.1}$$

Simply stated, the predicted execution time of an algorithm is the sum of the computational time and the communication time minus the overlap. This overlap is represented as a percentage (called ρ) or the smaller of the two components: communications and computation. Note that the overlap can be no greater than whichever component is the smallest (thus the use of the minimum operator in the equation). An overlap of 100 percent corresponds to either computation completely overlapping communication (with additional computation to spare) or communication completely overlapping computation (with communication left over).

The computational models were created by combining a set of basic building blocks built in Mathcad and shown in Table 3.2. These functions are derived from blocks of operations in the conventional beamforming algorithms. Each function is a representation of the number of floating-point operations required to complete the particular task in the beamformer. The FFT beamformers use all six of these functions while the time domain only uses $t_{\text{matrix_sum}}$, $t_{\text{window_mult}}$, $t_{\text{find_energy}}$ and $t_{\text{pick_angle}}$ in the calculations. The $t_{\text{matrix_sum}}$ represents the operations needed to reduce a data matrix into a single vector of angle results (via a sum operation). The $t_{\text{window_mult}}$ functions represents multiplication of a node's input data vector by its windowing factor. The operations for reducing the vector of angle results to scalar energy values for each angle is included in the $t_{\text{find_energy}}$ function. The $t_{\text{pick_angle}}$ function includes finding the angle of greatest signal energy. The $t_{\text{steering_mult}}$ function is used only in FFT beamformers and covers the operations required for applying a phase shift to the complex Fourier data before applying the matrix summation. Lastly, the $t_{\text{FFT_time}}$ function accounts for the transform of the incoming data vector into the frequency domain in FFT beamformers.

$t_{\text{matrix_sum}}(S, N, M) := S \cdot N \cdot (M - 1)$	$t_{\text{window_mult}}(M, N, \beta) := M \cdot N \cdot \beta$	$t_{\text{pick_angle}}(S) := 4 \cdot S$
$t_{\text{FFT_time}}(N) := 6 \cdot N \cdot \frac{\log(N)}{\log(2)} - N$	$t_{\text{steering_mult}}(S, N, M) := 7 \cdot S + 5 \cdot S \cdot N \cdot M$	$t_{\text{find_energy}}(S, N) := 2 \cdot S \cdot N$

M - Number of Nodes
N - Number of Samples in Sequence
S - Number of Steering Angles
 β - Reduction in Data by Non-linear Techniques

Table 3.2 – Computational Building Blocks Used in Prediction Models

These building blocks were then combined to create models (shown in Table 3.3) with differing degrees of parallelism (DOP) using the decomposition techniques previously described for the algorithms, including control and data parallelism. The table includes both parallel frequency-domain and time-domain conventional beamforming algorithms as well as non-linear optimized time-domain algorithms. Note that the numbers of operations are calculated on a per-node basis. The complexity of the time-domain algorithms is less than that of the frequency domain algorithms since they do not need a pre-processing stage (i.e. FFT), and they lack the complex multiplication found in the frequency domain. However, the conventional time-domain algorithms require a much larger sample space, especially if the number of nodes, N , and the number of steering directions, S , is large. The optimized non-linear time-domain techniques decrease the size of this sample space to that of the frequency domain. Figure 3.9(a) plots the computational requirements for frequency-domain algorithms (including both the unidirectional

and full-capability algorithms). Figure 3.9(b) shows the time-domain algorithms and the full-capability frequency-domain algorithms. Notice the scale of this figure has been changed so that more detail can be shown. Lastly, Figure 3.9(c) shows only the time-domain algorithms compared to one another. Note that in each of these plots, the abscissa is the number of nodes (M), and the ordinate is the worst-case number of floating-point operations per node per iteration. Also note that MPNF5 refers to the non-pipelined medium-grain full-capability FFT beamformer, and CPNF4 refers to the coarse-grain full-capability FFT beamformer. The coarse-grain and medium-grain unidirectional algorithms are labeled CPUNF and MPUNF, respectively. The pipelined programs use a parameter γ , which is a function of the number of nodes, to indicate the amount of overlap of pipeline stages. Lastly, note that $t_{\text{sum_and_IFFT}}$ is the sum of the operations in a matrix sum and an FFT, $t_{\text{FFT_and_window}}$ is the sum of $t_{\text{FFT_time}}$ and $t_{\text{window_mult}}$ for one vector of data, $t_{\text{freq_adj}}$ includes the operations required in the optimized time-domain algorithms for finding the base cycle of the signal, and α represents the fraction of iterations in which this cycle check is done.

$\text{Ops}_{\text{FFT_Sequential}}(S, N, M) := t_{\text{FFT_and_Window}}(N) \cdot M + t_{\text{steering_mult}}(S, N, M) + t_{\text{sum_and_IFFT}}(S, N, M) + t_{\text{find_energy}}(S, N) + t_{\text{pick_angle}}(S)$
$\text{Ops}_{\text{MPNF5}}(S, N, M) := t_{\text{FFT_and_Window}}(N) + t_{\text{steering_mult}}\left(\left\lceil \frac{S}{M} \right\rceil, N, M\right) + t_{\text{sum_and_IFFT}}\left(\left\lceil \frac{S}{M} \right\rceil, N, M\right) + t_{\text{find_energy}}\left(\left\lceil \frac{S}{M} \right\rceil, N\right) + t_{\text{pick_angle}}(S)$
$\text{Ops}_{\text{CPNF4}}(S, N, M, \gamma) := \left(\frac{1}{1 - \gamma(M)} t_{\text{FFT_and_Window}}(N) + t_{\text{steering_mult}}(S, N, M) + t_{\text{sum_and_IFFT}}(S, N, M) + t_{\text{find_energy}}(S, N) \right) (1 - \gamma(M)) + t_{\text{pick_angle}}(S)$
$\text{Ops}_{\text{MPUNF}}(S, N, M) := t_{\text{FFT_and_Window}}(N) + 8S + 5NS + t_{\text{sum_and_IFFT}}(S, N, M) + t_{\text{find_energy}}(S, N) + t_{\text{pick_angle}}(S)$
$\text{Ops}_{\text{CPUNF}}(S, N, M, \gamma) := t_{\text{FFT_and_Window}}(N) + t_{\text{steering_mult}}(S, N, M) + t_{\text{sum_and_IFFT}}(S, N, M) + t_{\text{find_energy}}(S, N) + t_{\text{pick_angle}}(S)$
$\text{Ops}_{\text{butterfly_conventional}}(S, N, M) := \frac{t_{\text{matrix_sum}}(S, N, M) + t_{\text{window_mult}}(M, N)}{M} + t_{\text{find_energy}}(S, N) + t_{\text{pick_angle}}(S)$
$\text{Ops}_{\text{butterfly_optimized}}(S, N, M) := \frac{t_{\text{matrix_sum}}(S, N, \beta(M), M) + \beta(M) \cdot t_{\text{window_mult}}(M, N)}{M} + t_{\text{find_energy}}(S, N) + t_{\text{pick_angle}}(S) + \alpha \cdot t_{\text{freq_adj}}(N)$
$\text{Ops}_{\text{pipelined_conventional}}(S, N, M) := (t_{\text{find_energy}}(S, N) + t_{\text{matrix_sum}}(S, N, M) + t_{\text{window_mult}}(M, N) + t_{\text{pick_angle}}(S)) (1 - \gamma(M))$
$\text{Ops}_{\text{pipelined_optimized}}(S, N, M) := (t_{\text{find_energy}}(S, N) + t_{\text{matrix_sum}}(S, N, \beta(M), M) + \beta(M) \cdot t_{\text{window_mult}}(M, N) + t_{\text{pick_angle}}(S)) (1 - \gamma(M)) + \alpha \cdot t_{\text{freq_adj}}(N)$
$\text{Ops}_{\text{GDS_conventional}}(S, N, M) := \frac{t_{\text{find_energy}}(S, N) + t_{\text{matrix_sum}}(S, N, M) + t_{\text{window_mult}}(M, N)}{M} + t_{\text{pick_angle}}(S)$
$\text{Ops}_{\text{GDS_optimized}}(S, N, M) := \frac{t_{\text{find_energy}}(S, N) + t_{\text{matrix_sum}}(S, N, \beta(M), M) + \beta(M) \cdot t_{\text{window_mult}}(M, N)}{M} + t_{\text{pick_angle}}(S) + \alpha \cdot t_{\text{freq_adj}}(N)$

Table 3.3 - Parallel Decomposition Equations

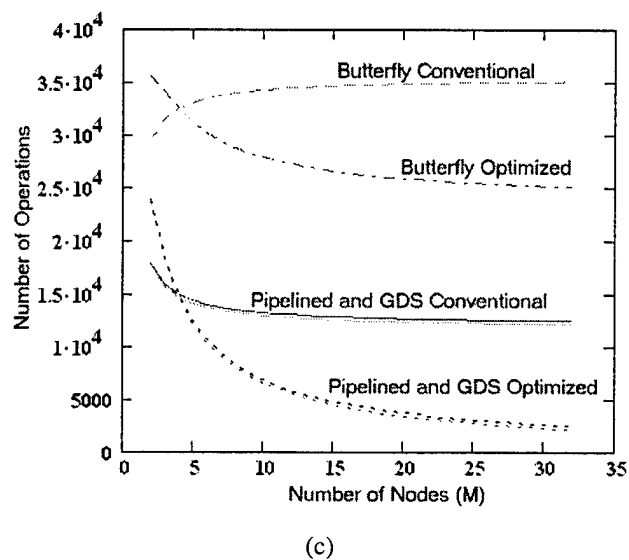
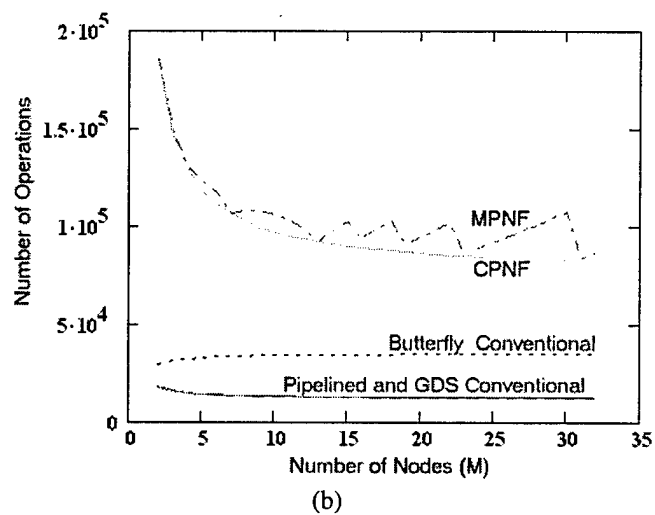
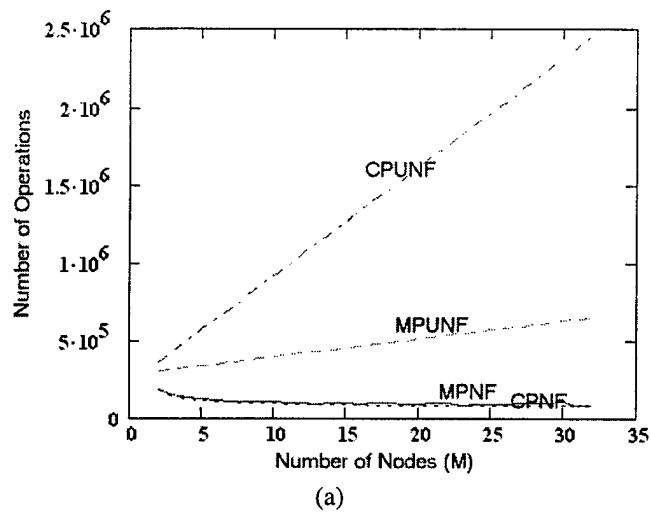


Figure 3.9 – Floating-point Operations per Iteration for Parallel Beamformers

The communication model was set up similarly to the computational models except that there was only one building block. This function (shown below in Table 3.4) has a number of inputs, which allows it to coarsely describe any network. The network aggregate factors describe the approximate degree to which two proposed network models (register insertion ring and bidirectional insertion array) utilize the network. On the average, the ring (as shown from the formula) will support two transactions at any time. On the other hand, the bidirectional array will support about two transactions at any given time for small networks (e.g. four nodes or less) and will support four transactions for large networks.

Assumptions: - broadcast capable - fully network connected - no contention hit - connectionless (no call setup) - all links evenly loaded - no acknowledgements		ε - overhead in floats per packet δ - number of divisions of transmitted data κ - size of message per iteration (in floats) ψ - network's aggregate bandwidth factor
$t_{\text{comm}}(\kappa, \delta, \varepsilon, \psi) = \frac{\delta \cdot \left(\frac{\kappa}{\delta} + \varepsilon \right)}{\psi}$	$\psi_{\text{ring}}(M) = \frac{M}{\left(\frac{M}{2} \right)}$	$\psi_{\text{bidirectional}}(M) = 4 - \frac{4}{M}$

Table 3.4 - Communication Functions

The overall communication cost of an algorithm was then calculated by summing the communication cost of all the transactions required by the particular algorithm. This procedure is shown below in Table 3.5. Note that the table shows the communication only over a register insertion ring (since the ψ_{ring} factor is used instead of the $\psi_{\text{bidirectional}}$ factor). The same method applies for the bidirectional array except the aggregate bandwidth factor for the bidirectional array is substituted. The communication cost for the conventional time-domain algorithms is much greater than the frequency-domain algorithms. This result is due to the large sample space required of the time domain. The optimized time-domain algorithms take advantage of much lower communication complexity through circular shifting. As a result, the optimized time-domain and frequency-domain algorithms have similar communication complexity patterns while the conventional time-domain algorithms suffer from large communication latencies.

$\text{Comm_ring_MPNFS}(M, N, S) := t_{\text{comm}}(M \cdot N \cdot M, \kappa, \psi_{\text{ring}}(M)) + t_{\text{comm}}\left(S - \text{floor}\left(\frac{S}{M}\right), M - 1, \kappa, \psi_{\text{ring}}(M)\right)$
$\text{Comm_ring_CPNFS}(M, N, S) := t_{\text{comm}}\left[N \cdot (M - 1), M - 1, \kappa, \psi_{\text{ring}}(M)\right]$
$\text{Comm_ring_MPUNF}(M, N, S) := t_{\text{comm}}\left[S \cdot N \cdot (M - 1), M - 1, \kappa, \psi_{\text{ring}}(M)\right]$
$\text{Comm_ring_CPUNF}(M, N, S) := t_{\text{comm}}\left[N \cdot (M - 1), M - 1, \kappa, \psi_{\text{ring}}(M)\right]$
$\text{Comm_ring_butterfly_conventional}(M, N, S) := t_{\text{comm}}(S \cdot M \cdot N, \kappa, \psi_{\text{ring}}(M)) + t_{\text{comm}}(S \cdot N \cdot M, \kappa, \psi_{\text{ring}}(M))$
$\text{Comm_ring_butterfly_optimized}(M, N, S) := t_{\text{comm}}(S \cdot M \cdot N \cdot \beta(M), \kappa, \psi_{\text{ring}}(M)) + t_{\text{comm}}(S \cdot N \cdot M, \kappa, \psi_{\text{ring}}(M)) + t_{\text{comm}}(16 \cdot \alpha \cdot M, \kappa, \psi_{\text{ring}}(M))$
$\text{Comm_ring_pipelined_conventional}(M, N, S) := t_{\text{comm}}\left[M \cdot \left(\text{ceil}\left(\frac{S}{2}\right) \cdot M + N\right), M, \kappa, \psi_{\text{ring}}(M)\right]$
$\text{Comm_ring_pipelined_optimized}(M, N, S) := t_{\text{comm}}(M \cdot N \cdot \beta(M), \kappa, \psi_{\text{ring}}(M)) + t_{\text{comm}}(S \cdot M, \kappa, \psi_{\text{ring}}(M)) + t_{\text{comm}}(16 \cdot \alpha \cdot M, \kappa, \psi_{\text{ring}}(M))$
$\text{Comm_ring_GDS_conventional}(M, N, S) := t_{\text{comm}}\left[M \cdot \left(\text{ceil}\left(\frac{S}{2}\right) \cdot M + N\right), M, \kappa, \psi_{\text{ring}}(M)\right] + t_{\text{comm}}(S \cdot M, \kappa, \psi_{\text{ring}}(M))$
$\text{Comm_ring_GDS_optimized}(M, N, S) := t_{\text{comm}}(M \cdot N \cdot \beta(M), \kappa, \psi_{\text{ring}}(M)) + t_{\text{comm}}(S \cdot M, \kappa, \psi_{\text{ring}}(M)) + t_{\text{comm}}(16 \cdot \alpha \cdot M, \kappa, \psi_{\text{ring}}(M))$

Table 3.5 - Communication Cost Equations

Comparing the communication of the four FFT decompositions, the equations predict that the non-pipelined medium-grain full-capability (MPNFS) program, the medium-grain unidirectional (MPUNF) program, and the coarse-grain full-capability (CPNFS) program will fare the best, while the coarse-grain unidirectional (CPUNF) program will do poorly. Figure 3.10(a), below, shows a Mathcad generated plot of the four FFT algorithms. Due to the poor performance of the FFT

unidirectional versions in both computation and communication, the remainder of this section concentrates on the two best frequency decompositions: MPNF and CPNF. The same situation is true for the time-domain butterfly algorithm, which does not fare as well as the pipelined and GDS techniques. Figure 3.10(b) below shows the communication of the time-domain beamformers where it can be seen that the pipelined and GDS algorithms perform very similarly while the butterfly technique, as discussed in Appendix B, has higher complexity. Figure 3.10(c) shows the communication for the full-capability FFT beamformers (CPNF and non-pipelined MPNF) against the conventional time-domain decompositions. Lastly, Figure 3.10(d) shows the communication for the same FFT algorithms compared to the time-domain algorithms with non-linear optimization.

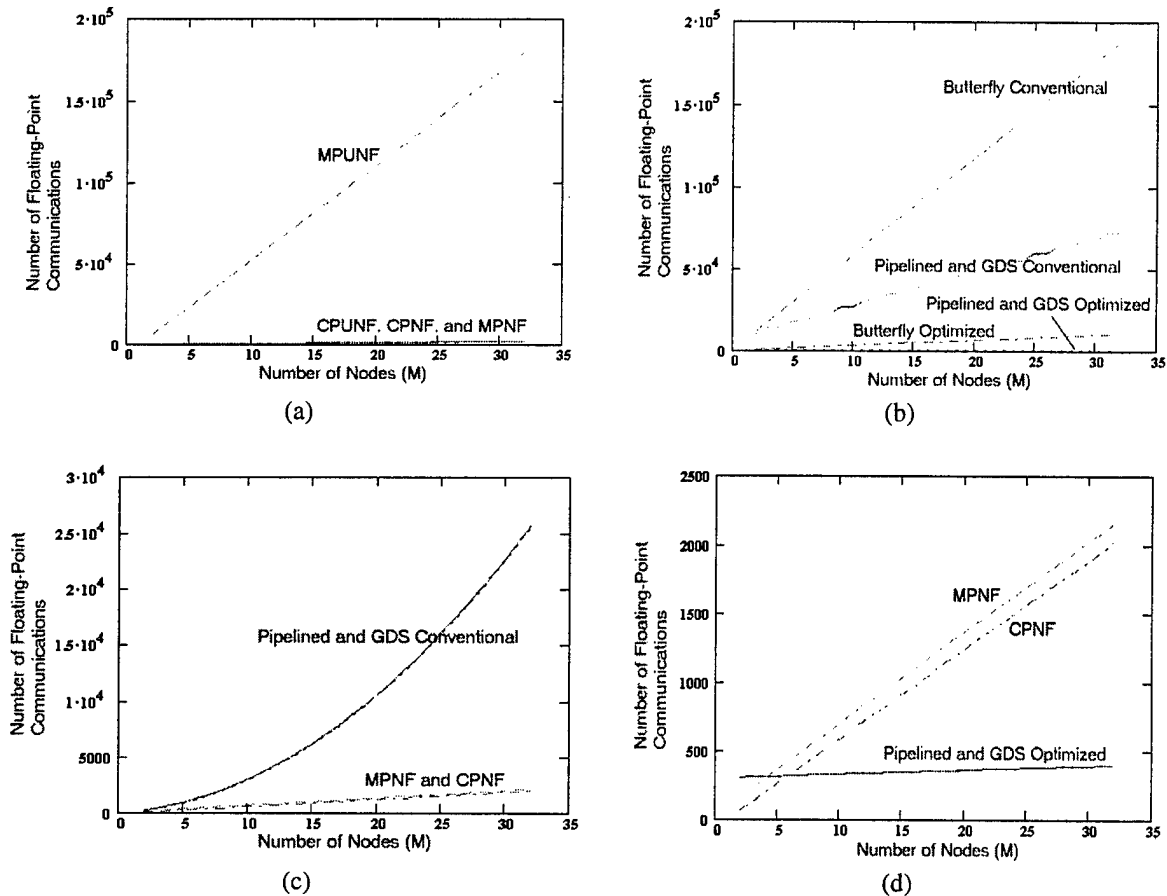


Figure 3.10 – Floating-Point Communications per Iteration for Parallel Beamformers

The total execution time was calculated using Equation 3.1 at the beginning of this subsection. Execution time was predicted by fixing a FLOPS (floating-point operations per second) rating to the network, a Mbps (megabits per second) rating to the processor, and an overlap factor of computation and communication to the variable ρ . For these results, the system was rated at 10 MFLOPS and 10Mbps with a ρ factor of 0.5. Figure 3.11(a), below, shows the coarse-grained FFT algorithm performing slightly better than the medium-grained non-pipelined FFT algorithm, because it does more computation and communication. For the time-domain algorithms, shown in Figure 3.11(b), the pipelined and GDS versions are equal in complexity. This result is no surprise since both the analytic equations for operations and communications can be algebraically manipulated to equal each other. This plot also shows the superiority of the

fractional-steering and circular-shifting methods (non-linearly optimized beamformers) to the non-optimized versions. The execution time of the time-domain beamformers is mostly due to the long communication, whereas, the bulk of the execution time for the frequency-domain algorithms is in the computation. Since the optimized time-domain methods work mostly to minimize the size of the sample space, the communication is minimized yielding tremendous improvements. Figure 3.11(c) combines Figure 3.11(a) and Figure 3.11(b) together. Notice that with a low number of nodes, the time-domain conventional algorithms perform very well, but as the number of nodes increases, the sample space gets so large that the communication requirements grow exponentially. With 10 nodes, FFT algorithms and non-optimized time-domain algorithms seem to perform equally well. Again, since the optimized time-domain decompositions reduce the size of the sample space, they outperform both conventional time-domain and frequency-domain algorithms for any number M . The last plot in this sequence shows the speedup of the frequency-domain algorithms and the conventional time-domain techniques over the sequential FFT beamformer. Note that it appears that the time-domain is achieving super-linear speedup in the range of 2-10 nodes, but this anomaly is simply because they are being compared to the frequency-domain sequential version. However, for more than 10 nodes, the performance of the time-domain programs falls off and shows poor speedup compared to the FFT sequential algorithm. This plot also reveals the low computational complexity of the time-domain beamformer but the large communication complexity when the number of nodes is large.

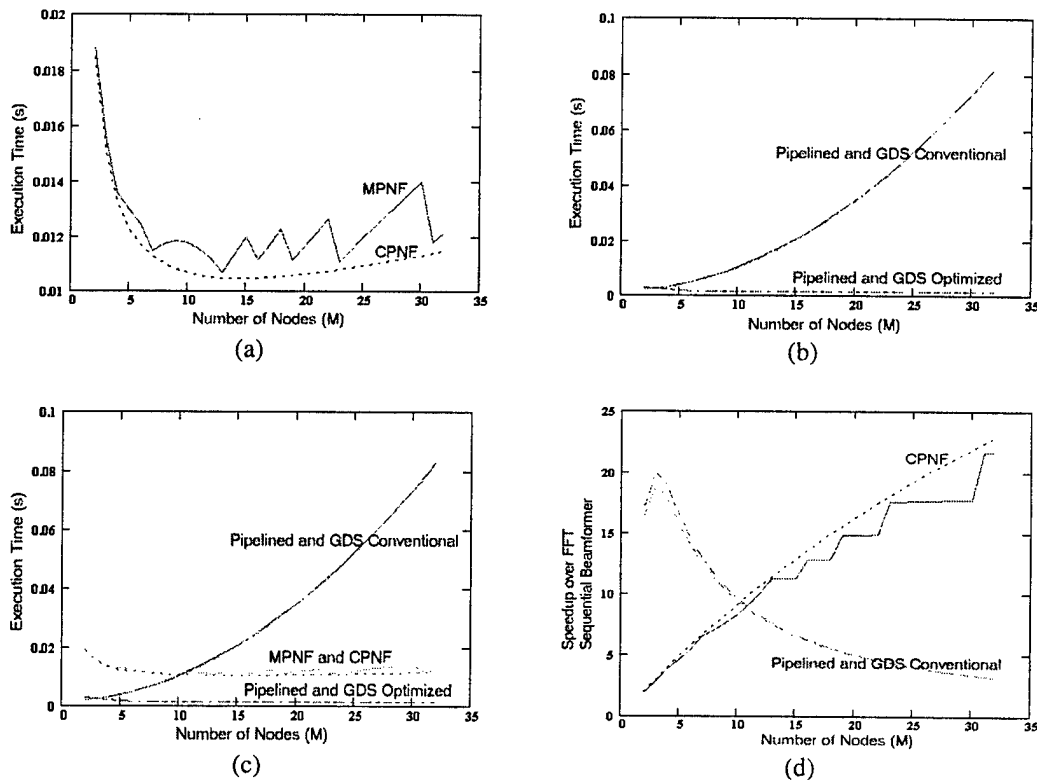


Figure 3.11 - Expected Execution times, FLOPS rating = 10M, Network Bandwidth = 10Mbps

Figures 3.12(a)-(f) show another set of predicted execution times for three FFT algorithms (the sequential, the medium-grain non-pipelined MPNF, and the coarse-grain CPNF) and the three non-linearly optimized time-domain algorithms (the GDS-TD implementation, the pipelined program, and the Butterfly). The execution time is plotted against two variables: number of

steering directions, S , and number of samples, N . The number of nodes, M , was fixed at 32 for these plots. The z-axis shows predicted execution time per iteration. Again, the system was rated at 10 MFLOPS and 10Mbps with a ρ factor of 0.5. Notice in all cases that the complexity of the problem increases linearly with N and linearly with S . The medium-grained FFT algorithm and coarse-grained FFT algorithm perform an order of magnitude better than the sequential algorithm while the pipelined time-domain and GDS time-domain algorithms perform two orders of magnitude better. The time-domain Butterfly algorithm, however, is only twice as fast as the sequential FFT beamformer and is outperformed by all other algorithms.

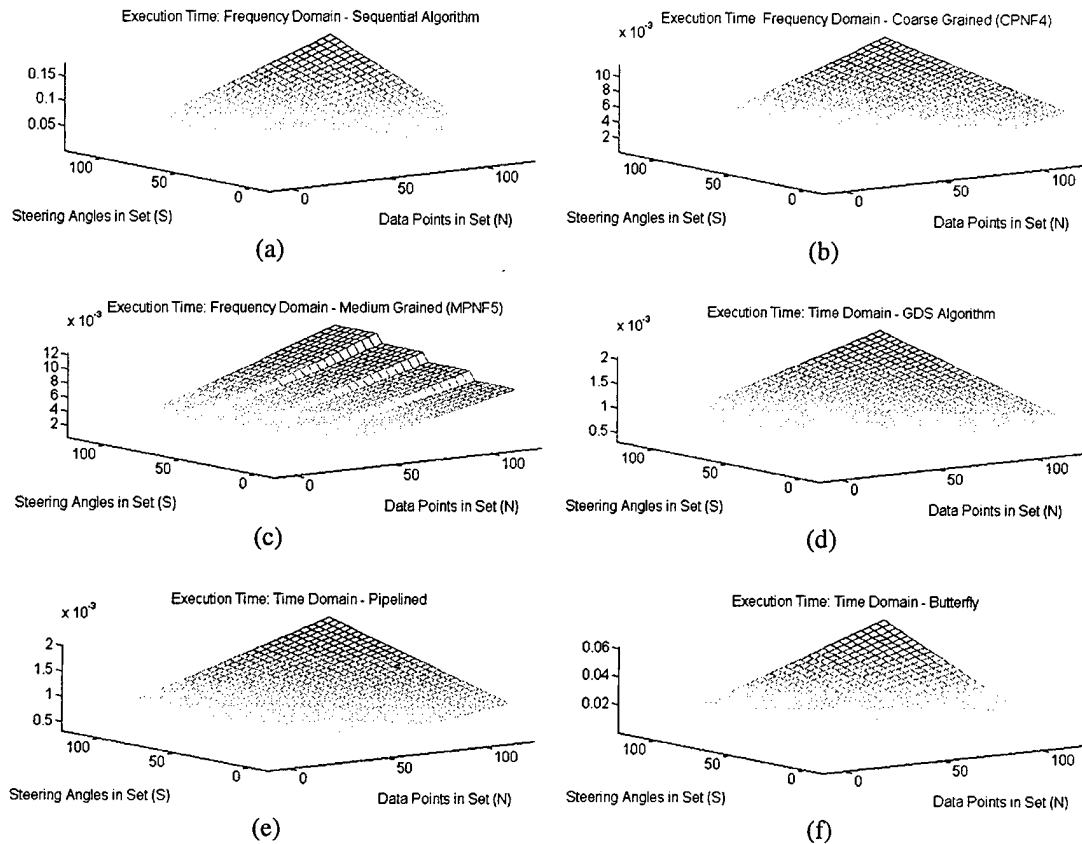


Figure 3.12 - Predicted Execution Times versus Number of Steering Directions and Number of Nodes

3.4. Performance Results

This section presents the performance of the parallel programs in terms of speedup over a baseline. Before comparing the algorithms in this section, it should be emphasized that the execution times collected depend heavily upon the speed of the processor, and the throughput and latency of the interconnection network. The testbed consisted of a cluster of 85-MHz SPARCstation-20 workstations connected by 155-Mbps (OC-3) ATM. The anticipated sonar array architecture will consist of slower processors and a lower throughput network.

For each program, the baseline is chosen so that the assumptions lead to valid conclusions. For example, all network-independent programs are compared against the network-independent baseline in which all dumb nodes send their raw data to the front-end intelligent node via one MPI call. The network-specific programs are compared against a unidirectional array of dumb nodes in which the nodes append raw data to a freight train that arrives at the front-end intelligent

node. In both versions of the baseline, the intelligent node executes the sequential program using data obtained from the dumb nodes, which do no computation on the data. The speedup numbers obtained for all the network-independent programs are shown Figure 3.13. The beamformer was set to steer every half-degree in these charts. Figure 3.14 shows the respective efficiencies measured as the ratio of speedup to number of nodes. Recall the following acronyms: the coarse-grain unidirectional programs are CPUF for the version with the rudimentary unidirectional network simulation and CPUNF for the version with no rudimentary network. CPRF, CPBF, and CPNF are the coarse-grain, full-capability FFT beamformers with rudimentary ring simulation, rudimentary bidirectional array simulation, and no rudimentary network, respectively. The same pattern holds true for the medium-grain counterparts except that the first letter of the acronym is "M" instead of "C."

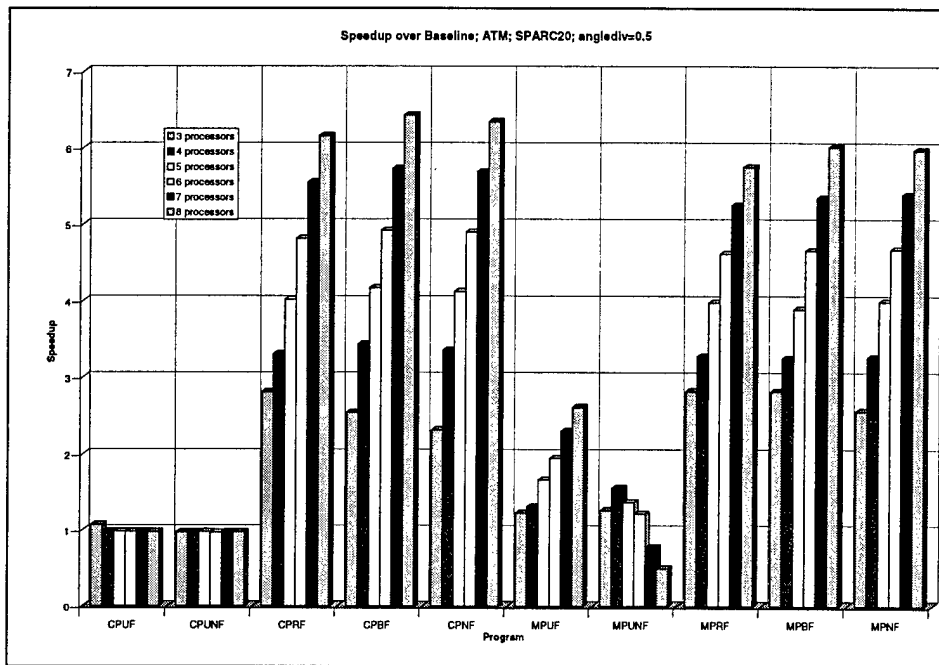


Figure 3.13 - Speedups versus Number of Nodes. The angle division set to 0.5 leads to 361 steering directions within the 180-degree space. The most notable features of this graph are the relative performance numbers, the invariant speedup for the coarse-grained unidirectional algorithms, and the inverse speedup for the medium-grain, unidirectional, network-independent algorithm.

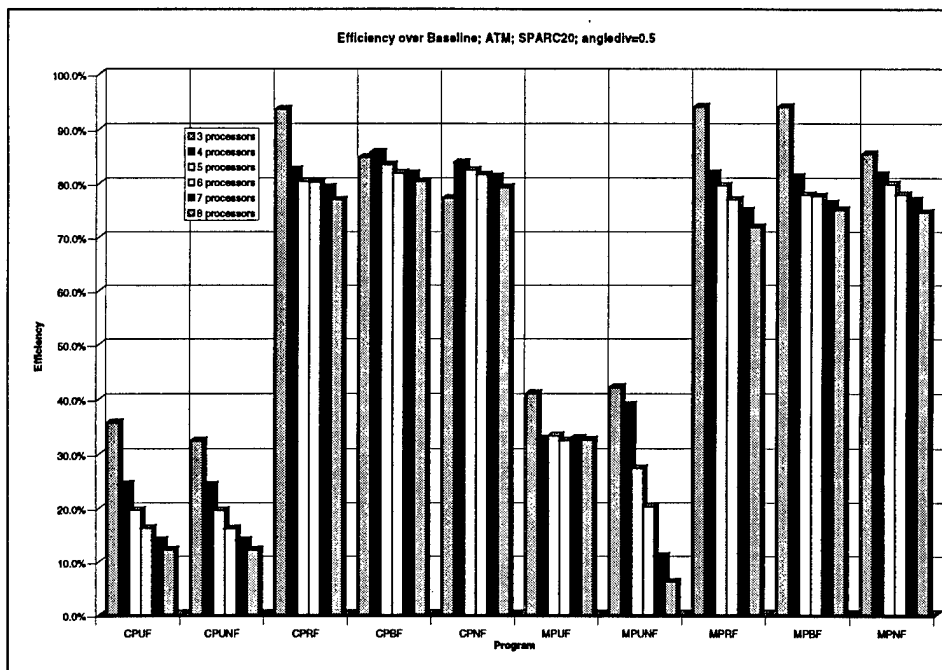


Figure 3.14 - Efficiencies of Programs versus Number of Nodes. This graph illustrates the ratio of speedup to number of nodes (i.e. the percentage of ideal speedup). For nearly all of the algorithms, the efficiency is decreasing, leading to the conclusion that such algorithms may encounter a scalability limit as more nodes are added. Simulation of larger arrays will help establish if such a limit exists.

As can be seen in the figures, the coarse-grain unidirectional algorithm does not provide any speedup over the baseline. In these algorithms, there is very little parallelism. In fact, the only difference between the baseline and these programs is the parallelization of the opening sequence (FFT and window factor multiplication) which was shown to represent a small portion of overall execution time. The coarse-grained algorithm for fully-connected networks improves upon the unidirectional algorithm's performance considerably. The algorithm scales near-linearly as more nodes are added, which can be seen by the slow decrease in efficiency. Speedups reach above 6 for 8 nodes, giving efficiencies of more than 70 percent. The medium-grained unidirectional algorithm performs better than the coarse-grained unidirectional algorithm due to the increased degree of parallelism. The medium-grained unidirectional network-specific program scales better than the coarse-grained unidirectional program but not as much as the coarse-grained fully-connected programs. It reaches speedups of around 2.5 for 8 nodes giving an efficiency of just over 30 percent. The medium-grain network-independent unidirectional program begins to perform poorly with many nodes. This result can be explained by realizing that the front-end node is being inundated with communication requests (all nodes send their steering-direction data directly to the front-end node via a single MPI call). Because this program sends considerably more data for each iteration than the coarse-grained unidirectional network-independent counterpart or the baseline, the ATM network goes into saturation and begins dropping cells requiring the packets to be re-sent. The medium-grained fully-connected algorithms perform with speedups just below those of the coarse-grained programs. This result is due to the increase in the amount of communication for the same amount of computation. Speedups reach just below 6 for 8 nodes resulting in efficiencies in the lower 70's. However, the medium-grained algorithms have the advantage over the coarse-grained algorithms of parallelism within an iteration. This fact means that the results of an iteration are available sooner in the medium-grained algorithms. In the coarse-grained algorithms, the results for an iteration are only available after the pipeline has been traversed.

Since the performance of the medium-grain algorithms depends on the number of steering directions, Figure 3.15 shows the speedups of the programs for three different steering-direction increments with an 8-node beamformer. The efficiencies for these speedups are shown in Figure 3.16.

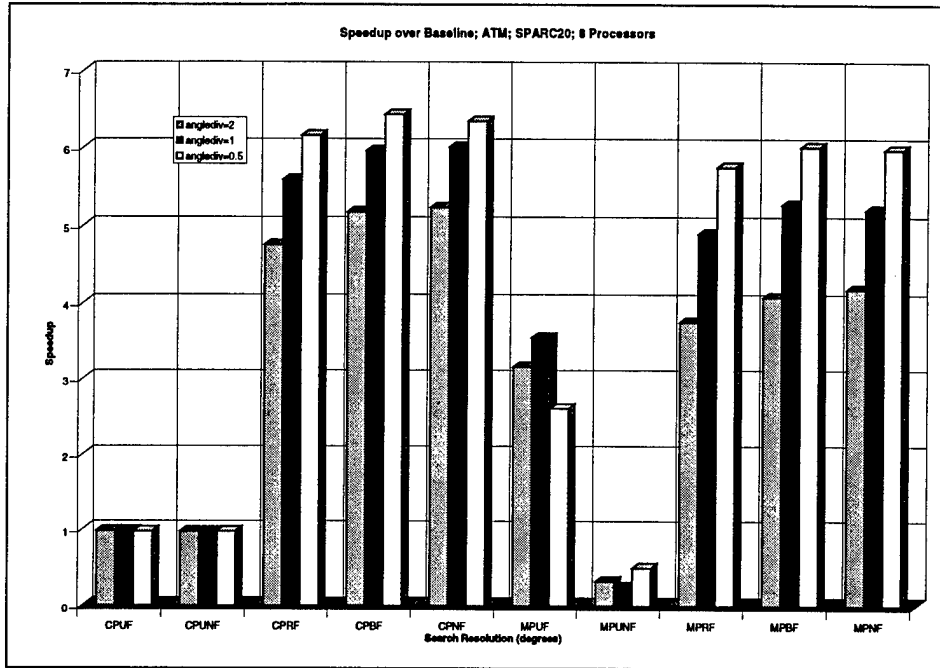


Figure 3.15 - Speedups versus Number of Steering Directions

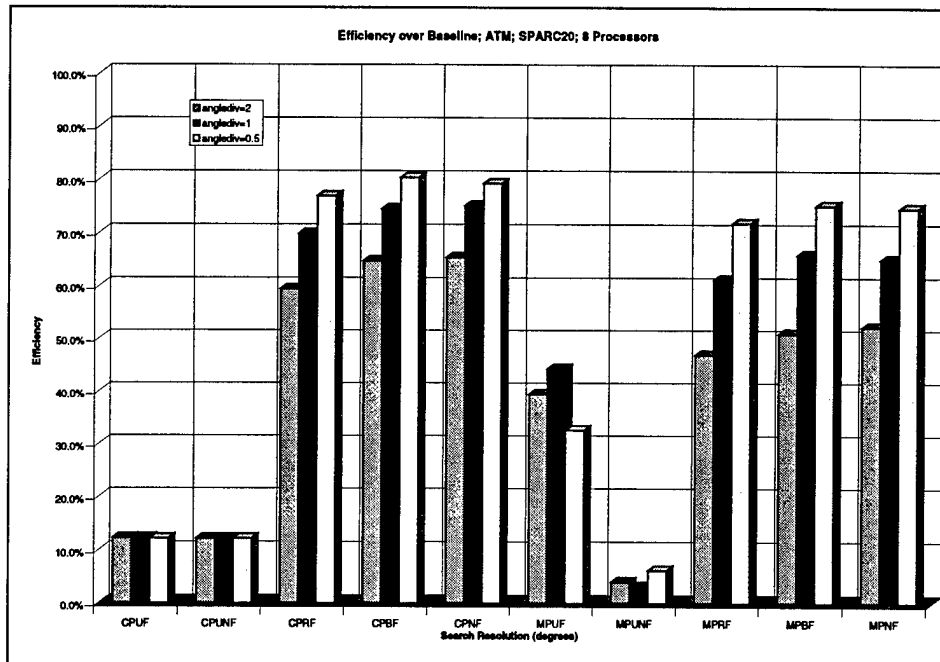


Figure 3.16 - Efficiencies versus Number of Steering Directions

The coarse-grained unidirectional performance remains poor for all steering-direction resolutions. The coarse-grained fully-connected algorithms improve their efficiency as steering directions are added due to the fact that no more communication is needed to do the additional steering directions. The medium-grained unidirectional program must balance the amount of computation with the amount of communication. As more steering directions are added, more communication is necessary. Furthermore, the granularity knob must be set so that this communication occurs in more pieces (so that the pieces are not so large as to overwhelm the MPI implementation and cause the implementation to use synchronous sending). The additional communication is not balanced by the additional computation of more steering directions as the resolution is increased to every half degree. Thus, the peak in performance is seen for a resolution of 1 degree. For the network-independent medium-grained unidirectional algorithm, the performance is poor across the board due to the network saturation. For the medium-grained fully-connected programs, the performance increases as steering directions are added, which is due to the fact that each node has more work to do (a larger share of steering directions) as more directions are added, but no more communication to complete.

3.5. Conclusions

The first steps completed in conventional beamforming for phase one of this project have been expanded upon and completed for phase two. Additional algorithms to improve upon the original parallel algorithms were created to either provide performance improvement or simply provide an alternative algorithm that may work better under different architectures. These programs were fully examined for overall execution time, speedup over baseline, etc. They were also broken down into component parts, each of which was investigated to determine where improvements could be made in coding the algorithm or changing the algorithm itself.

The results of this section have concluded that the coarse-grained programs provide the best raw execution time. The medium-grain programs follow close behind and offer a more robust parallelization. Either of the full-capability FFT beamformers will provide the desired performance for the autonomous sonar array.

Phase two of the project also involved implementing extensions to the time-domain beamformer so that it may better compete in execution time with the fast-performing FFT programs. Though no argument can be made that the time-domain program can outperform the FFT beamformer, the research into the time domain led to the GDS approach to real-time sonar array processing. This knowledge has turned out to be invaluable toward implementing the fault-tolerant aspects of the autonomous sonar array (as described in Appendix E).

4. Advanced Beamforming Algorithms

Split-aperture conventional beamforming is one of the latest beamforming techniques considered for decomposition over parallel processors. Because of the increase in complexity, the system has a higher degree of parallelism (DOP) and is thus likely to achieve better speedup over its sequential version than the CBF algorithms previously implemented. This section focuses on the fundamentals of split-aperture CBF and a few decompositions that are currently in experimentation.

Due to the characteristics of conventional beamforming algorithms and high computational cost, implementation of the algorithms requires high-speed and high-power systems. Using the Split-Aperture Conventional Beamforming (SA-CBF) algorithm this problem can be reduced. By using τ -interpolation and the phase-compensation method, the resolution of the beamforming output can be increased even with a small number of steering angles. Also the Smoothed COherent Transform (SCOT) and the phase transform make the beamforming plot more precise and accurate with less variance. The details of algorithm and computer simulation are given in Appendix C such as more in-depth treatment of the SA-CBF stages, equations, figures, data generation for computer simulation and complex data simulations.

4.1. The Split-Aperture Conventional Beamforming Algorithm

The SA-CBF algorithm is designed to work in the frequency domain. After an FFT converts the incoming data to the frequency domain, all the data processing is executed in the frequency domain until the post-processing stage. The computational cost of the FFT and the inverse FFT does not outweigh the advantages gained by using frequency-domain processing.

The nodes comprising the complete sonar array are logically divided into two sub-arrays. Each sub-array performs conventional frequency-domain beamforming using replica vectors on its own data independently. The two sub-array beamforming outputs are cross-correlated to detect the time delay of the signal for each steering angle. This cross-correlated data, with knowledge of the steering angles and several other parameters, will map to the final beamforming output. The individual steps of the overall process, shown in Figure 4.1, are explained in greater detail within the following subsections and Appendix C.

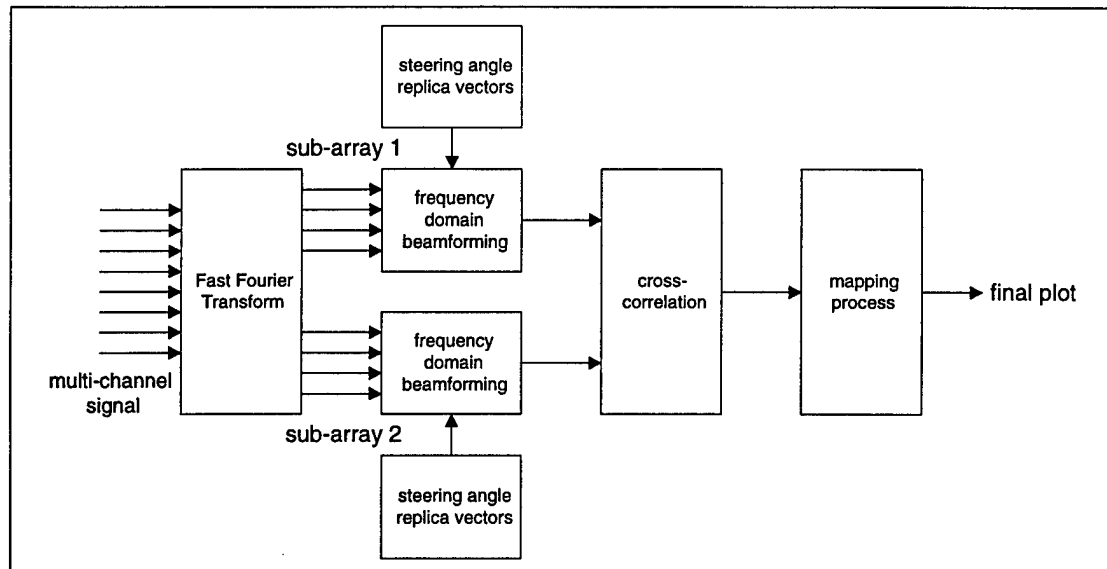


Figure 4.1 - Block Diagram for the Split-Aperture Conventional Beamforming Algorithm

The first stage in the above figure is the Fourier transform stage. If cross-correlation is implemented in the frequency domain without performing a sufficiently long FFT, an undesirable wrap-around effect will occur. This effect is a circular action that contaminates the original output of the correlation. To avoid the wrap-around effect in the cross-correlation stage, the input data needs to be zero-padded before the signal is passed to the FFT. The number of zeros depends on the length of the input data.

With lengthy FFT sets, the computational cost of the algorithm severely increases. There are two methods that can be employed in the FFT stage to reduce this side effect. One is frequency-bin averaging, and the other is ignoring the wrap-around effect.

The second stage is frequency-domain beamforming in SA-CBF, which is basically the same as single-aperture frequency-domain beamforming except the phase centers of each sub-array are considered. In this stage, the plane wave replica vector multiplies with the input transformed data; hence, the input data of the node is steered to the specific direction in relation to the phase center. The phase center is the reference point used to calculate the cross-correlation.

Unlike the single-aperture beamforming algorithm, the split-aperture algorithm does not need to steer at every individual desired angle. The cross-correlation causes some redundant information between the adjacent steering angles so each angle generates a range of the time delay plot.

The third stage is the cross-correlation stage. Cross-correlation is used to detect the time delay between the two phase centers. The maximum peak of the cross-correlation indicates the time delay of the two beamformer outputs.

We are only interested in the small number of angles or time delays in the cross-correlation adjacent to the beamforming angle of the sub-array since sub-array nodes are already steered to the specific direction. Beyond this range of angles, the cross-correlation is not correct so that portion of the cross-correlation must be eliminated from the final output to maintain correctness.

Before the inverse FFT is applied to obtain the cross-correlation as function of time delay, the Smoothed COherent Transform (SCOT) is performed for more distinct shaping of the cross-correlation. Fundamentally, SCOT sets the magnitude portion of the cross-correlation in the frequency domain to 1. This procedure results in a better resolution in the output plot because a

wide bandwidth in the frequency domain corresponds to a shaper image in the time domain. This spectral whitening is accomplished either by taking the instantaneous magnitude of the cross-correlation in frequency or a running average of the magnitude, which is calculated with the previous magnitude of the data. Sometimes, a spectral window is used to acquire the smooth beamforming plot.

Finally, a normalizing factor, which is the integral of the magnitude coherence, is used. This factor divides the SCOT result and ensures a normalized function with magnitude less than unity in time domain.

The fourth and final stage involves mapping the cross-correlations to the output. There are two methods of mapping the two cross-correlations to the final output correlation. The first method is to apply the weight function to the individual beam correlations with some overlap of neighbor correlations, and then add up all the cross-correlation values time-by-time. The weight function center is placed at the steering angle of the sub-array so that we take only the accurate values from each correlation. The other method is τ -interpolation, which involves taking only a range of cross-correlation values and then working with raised-cosine weights to figure out the interpolated angle from the two adjacent steered angles. In this case, only some range of cross-correlation values are required to calculate the final beamforming plot. For the output angle, two correlation functions will be evaluated in the closest beam pair

4.2. Computer Simulation

In this section, the split-aperture conventional beamforming algorithm is demonstrated graphically. Figure 4.2 shows the flowchart of the SA-CBF based on matrix operations.

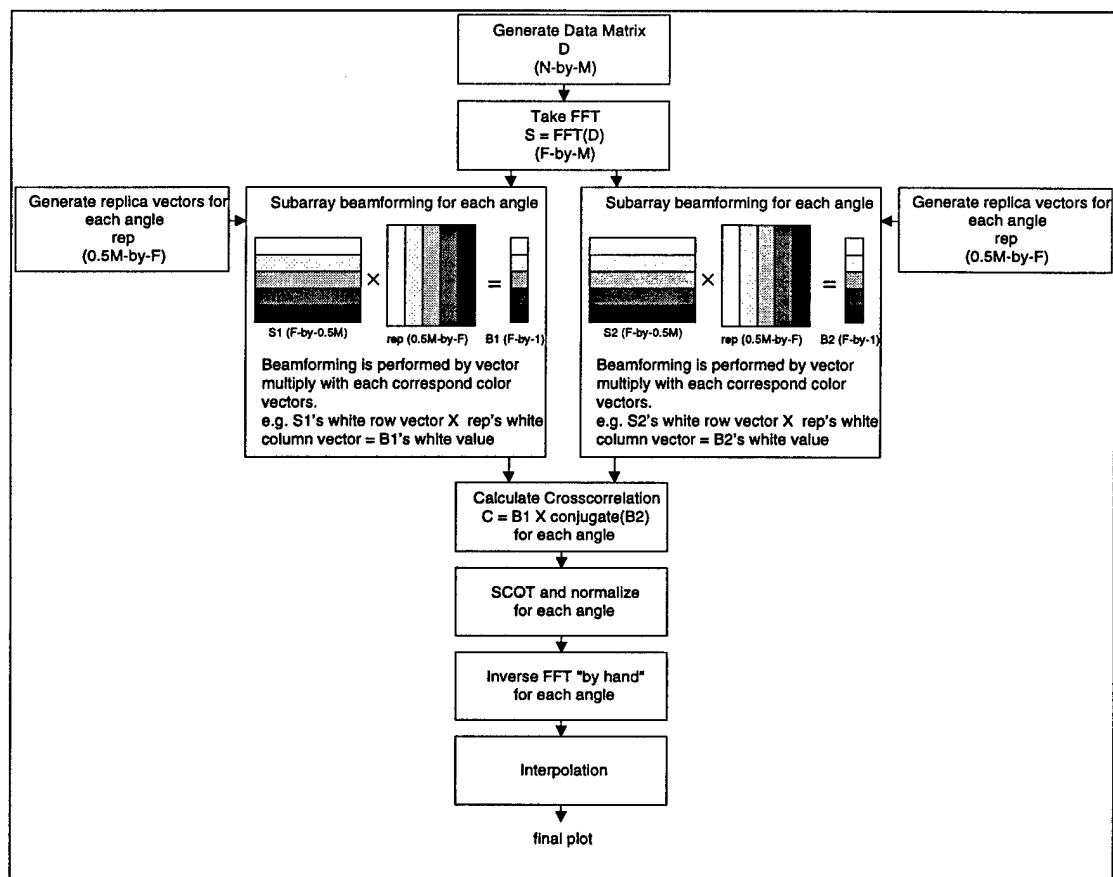


Figure 4.2 – Flowchart of the SA-CBF. M : number of nodes, N : number of samples and F : number of frequency bins

We first specify the array geometry and signal processing parameters. The geometry topology used in this simulation is the uniform linear array of nodes, which means the nodes are located with equal distance from each other. The reader is referred to Appendix C for parameters used in this simulation.

4.2.1. Intermediate Data Products.

According to Figure 4.2, there are several stages between generating the data and obtaining the final plot. By looking at the intermediate results of the SA-CBF, we can observe how the algorithm works and analyze it more easily. In the following examples of intermediate results, we are looking in the 47.8° direction. Figure 4.3 shows the results of each sub-array's beamform in that direction and of the cross-correlation between sub-arrays. The outputs of the sub-array beamformers keep the shape of the input signal, and the magnitude of the output is maximum because the signal is arriving from the steering direction. The cross-correlation of these two beamformer outputs detects the exact time difference between the two plots for sub-array results. However, we need only some range of time delay that unifies the sub-arrays (indicated by the arrow) mapped to the composite output display.

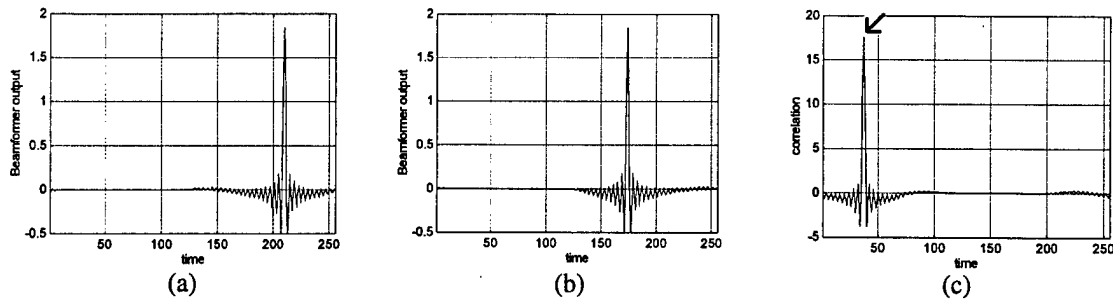


Figure 4.3 - Steered to the data incoming angle (47.8°); (a) subarray1 beamforming output; (b) subarray2 beamforming output; and (c) cross-correlation without SCOT and normalization.

If we consider another angle like -90° , Figure 4.4 will result. There is less power and magnitude in the sub-array beamformer outputs. Also, the shape is not same as the input signal. The correlation output does not have a prominent value, so it is hard to detect the time delay of the two sub-arrays for the signal. The arrow points to the range of the values that are mapped to the final output plot.

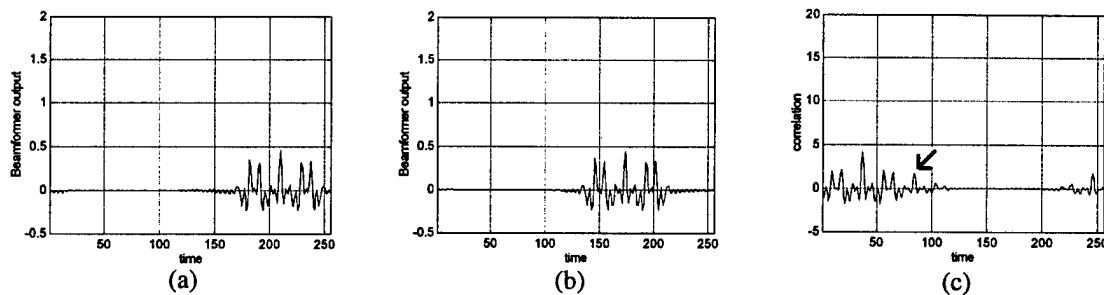


Figure 4.4 - Steered to the -90° (data angle is 47.8°); (a) subarray1 beamforming output; (b) subarray2 beamforming output; and (c) cross-correlation without SCOT and normalization.

In the next stage, SCOT and normalization is used. The normalization assists in drawing an independent plot for signal power. No matter how much signal power is delivered to the system, the final plot will stay with same maximum value. It is only sensitive to the direction of the incoming signal and noise. The normalized cross-correlation is plotted in Figure 4.5a and the normalized SCOT is plotted in Figure 4.5b. Notice the relative ease with which the time delay can be determined from the normalized SCOT plot. Also note that the SA-CBF output has more contrast with SCOT (as seen from the reduction of the peak pointed to by the arrow).

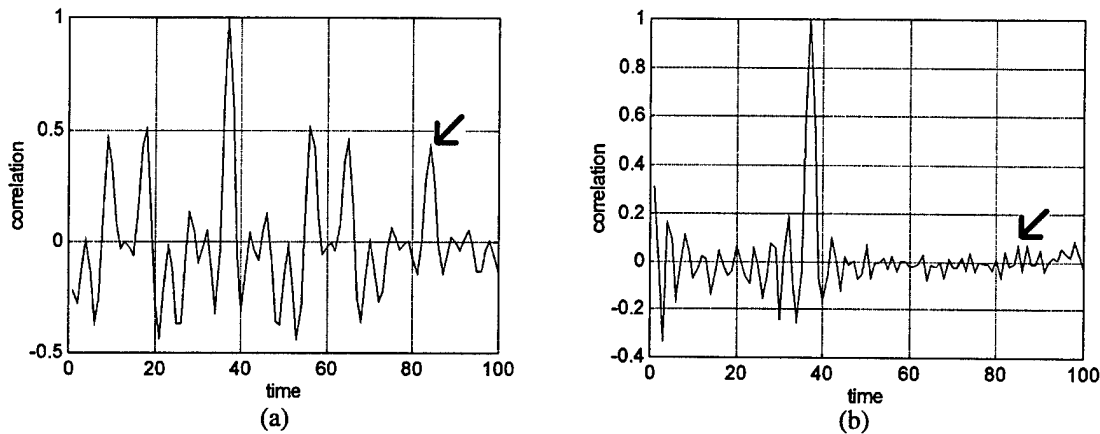


Figure 4.5 - Effect of SCOT and Normalization: (a) Figure 4.4c redone with normalization and (b) Figure 4.4c redone with SCOT and normalization.

The final SA-CBF output is shown in Figure 4.6 below with 3 output angles for every (sub-array) steering angle.

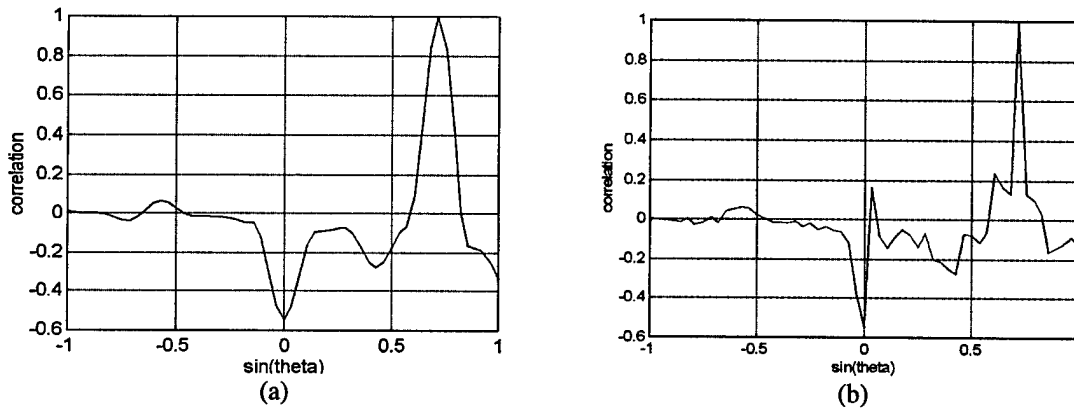


Figure 4.6 – SA-CBF output with 15 sub-array steering angles and 57 output angles: (a) with linear interpolation and (b) with τ -interpolation.

4.3. Performance Analysis

The performance of a beamforming algorithm can be determined in several ways (depending on the desired viewpoint). All of the performance factors are related to each other in some sense, so it is difficult to find out the optimal trade-off point. If we think about the implementation of the beamforming algorithm, the choice of the parameters is more complex. In this section of the paper, we compare the major performance factors between the SA-CBF and CBF.

4.3.1. SA-CBF vs. CBF

The conventional beamforming (CBF) algorithm is simply represented by the delay-and-sum model, which can be implemented in either the time domain or the frequency domain. The major difference between the single-aperture, delay-and-sum CBF and the split-aperture CBF is that

CBF has one phase center and SA-CBF has two phase centers. CBF attempts to add up the signals from all nodes to match one phase center via some delays, but SA-CBF splits the input signal into two parts via beamforming with two phase centers separately and then cross-correlates the beamformed signals.

For the sake of argument, we will consider in the comparison only the parameters for resolution and steering angle. It is obvious that more steering angles correspond to higher resolution but also higher computational cost. For the same number of steering angles, the SA-CBF has better resolution (see Figure 4.6b) than CBF because the SA-CBF is using some additional information at the cross-correlation stage. The visual resolution of the CBF can be increased by linear interpolation, but this is not real resolution. SA-CBF increases the real resolution by steering the two sub-arrays at the cross-correlation stage (see Figure C.3). The τ -interpolation technique helps to increase the number of output steering directions by interpolating values near the sub-array steering angles (which are the angles for which cross-correlation is exact and needs no interpolation).

4.3.2. Parallel Decompositions of SA-CBF

Decomposing the SA-CBF is a non-trivial task, not because of a limited degree of parallelism (DOP) but rather because of the many possibilities available for parallelizing the algorithm. The reason for the increased DOP is simply the increase in complexity of the SA-CBF algorithm. Figure 4.7 below shows the many added stages of SA-CBF along with a few proposed decompositions.

The plot on the left shows two techniques that have already been employed for conventional time and frequency-domain beamformers: that is, iteration decomposition and steering decomposition. Iteration decomposition is a coarse-grained algorithm that is a control-parallel approach. The advantages of this method are efficiency and a straightforward approach (which yields well to fault-tolerance). However, as is the case with the coarse-grain full-capability algorithm (described in Section 3), an iteration-parallel program is more likely to yield the longest latency for data from a particular iteration. Parallelism is achieved through pipelining.

Figure 4.7(a) also shows steering or angle decomposition. Steering decomposition is a data-parallel approach in which any one beamforming iteration is partitioned by steering angle and then accumulated into one solution in the end. Note, however, that because the angle-decomposition loop only affects a portion of the flowchart, a modular algorithm may be employed with angle decomposition as one of the modules. The portion of the flowchart outside this module is sequential code. By modularizing, these sequential bottlenecks may be parallelized independently of the parallelization implemented for the steering-angle calculation.

The decomposition technique shown in Figure 4.7(b) is a fully modularized fine-grained algorithm. This diagram shows the different modules and differentiates the type of parallelism employed in each. This fine-grained model is likely to yield the lowest result latency, but a coarse-grained model is likely to yield the best speedup. Low result latency is due to the data-parallel approach, in which many nodes split up the task. Whereas, in control parallelism, tasks remain intact and parallelism is achieved across multiple tasks. Thus, individual tasks do not finish as fast as tasks in the data-parallel approach.

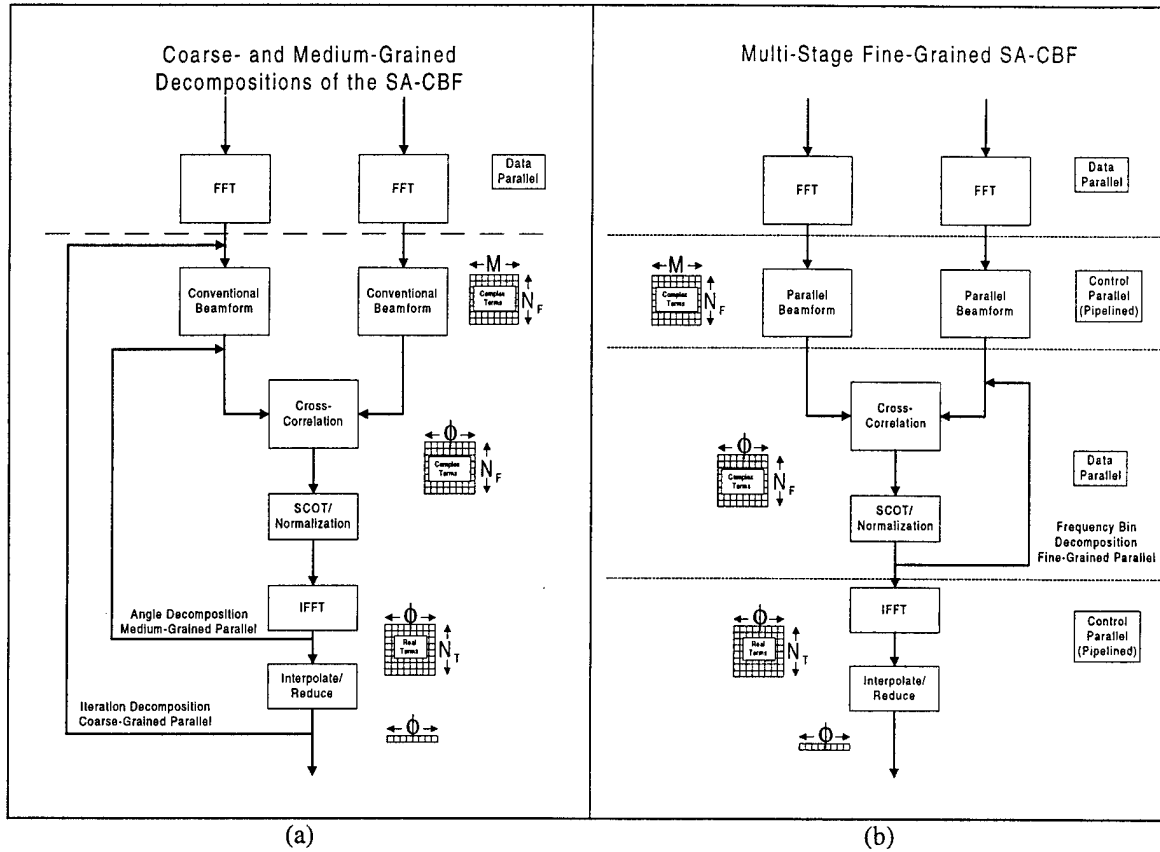


Figure 4.7 – Decompositions of the SA-CBF Algorithm.

4.4. ABF Description

So far, we have described the SA-CBF algorithm. CBF is a technique for localizing a target in a reasonably quiet environment. The problem with CBF is that the beamformer output is sensitive to the environment and the noise. Even if we apply a shading coefficient to each node, the beamformer still contains side lobes (for directions other than the desired steering direction) in the beam pattern as shown in Figure 4.8. When a strong signal comes from the direction of a side lobe, this unwanted signal power contributes to the beamformer output. It is impossible to remove all the side lobes of the beam pattern. We can reduce the size of side lobes or sharpen the main lobe by applying different windows to the nodes.

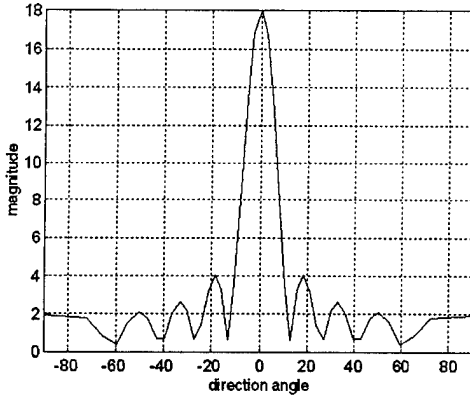


Figure 4.8 - Beam pattern of the SA-CBF with a uniform shading (no window applied)

Adaptive beamforming (ABF) takes into account the location of the nulls in the beam pattern and attempts to steer them at any unwanted noise. The replica vectors are manipulated in such a way that the background noise is minimized; therefore, nulls are steered at as many strong noise targets as possible. If the background noise during the experiment is about the same as during the time just prior to the experiment, then we can have a higher signal to noise ratio because the cost function or the performance surface of the signal keep the same shape, and the adaptive algorithms can easily find the optimal value of the coefficient. Otherwise, the information about the signal and background noise should be updated to calculate the new coefficient.

A problem in the ABF is that when strong background noise is located nearby the target direction, the ABF will steer nulls at the target, effectively making the target disappear. To avoid this problem, we need to be able to distinguish between the background noise and the desired signal by some method. One of the techniques is to give the beamformer some characteristics from both CBF and ABF. By placing in the system some parameter that controls the relative ratio of CBF use to ABF use, the beamformer can take advantage of both worlds.

ABF uses the Cross Spectral density Matrix (CSM) for calculating the optimal coefficients. CSM is defined as, $R = E[XX^*]$, where X is the vector of the Fourier components near some frequency from each node. Therefore, the size of the CSM should be M by M , where M is the number of nodes. The fact that the CSM is calculated for each frequency bin (or frequency range) means that there exists a third dimension (frequency) of the matrix.

The output power spectral density of the conventional beamformer is given in Equation 4.1, where v is the normalized replica vector for the specific frequency.

$$BF = \frac{1}{\Delta f} E[|v^* X|] = \frac{1}{\Delta f} E[(v^* X)(X^* v)] = \frac{1}{\Delta f} v^* E[XX^*] v = \frac{1}{\Delta f} v^* R v$$

Equation 4.1

The purpose of the ABF is to minimize this output power of the beamformer with some constraint function. This constraint function specifies that the beam pattern must not attenuate in the desired direction, but there is freedom in the choice of the null positions. The constraint function and the cost function to minimize are given in Equation 4.2, where w is the optimal weight for the ABF.

$$w^* v = v^* w = 1 \text{ (constraint function),} \quad BF = w^* R w \text{ (cost function)}$$

Equation 4.2

The weight vector, w , can be derived by using Lagrange Multipliers, which minimize the cost function within the constraint. The result of the Lagrange Multiplier is shown in Equation 4.3.

$$w = \frac{R^{-1}v}{v^* R^{-1}v}$$

Equation 4.3

To implement SA-ABF, the above calculated w coefficients are used instead of the original replica vectors.

Calculation of the CSM and inverse of the CSM are very computationally intensive, so it is desirable to reduce the computational cost by averaging and interpolating the frequency bins. The frequency averaging between adjacent bins may lose some signal information, so we need to find a compromise between resolution and computational cost. Figure 4.9 shows the procedure for finding the ABF coefficients using the frequency-averaging technique.

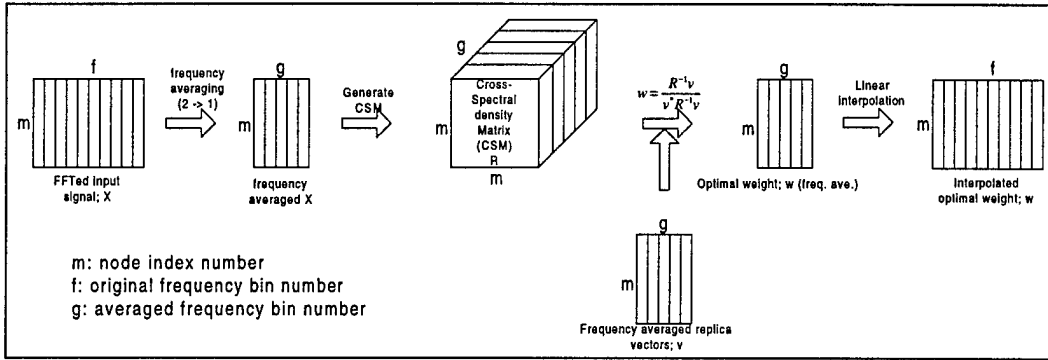


Figure 4.9 – Diagram for the ABF weight with frequency averaging.

Figure 4.2 can be redrawn (Figure 4.10 below) to represent the SA-ABF algorithm. Notice that the operations in Figure 4.9 are the operations that occur at the weight-generation stage in Figure 4.10.

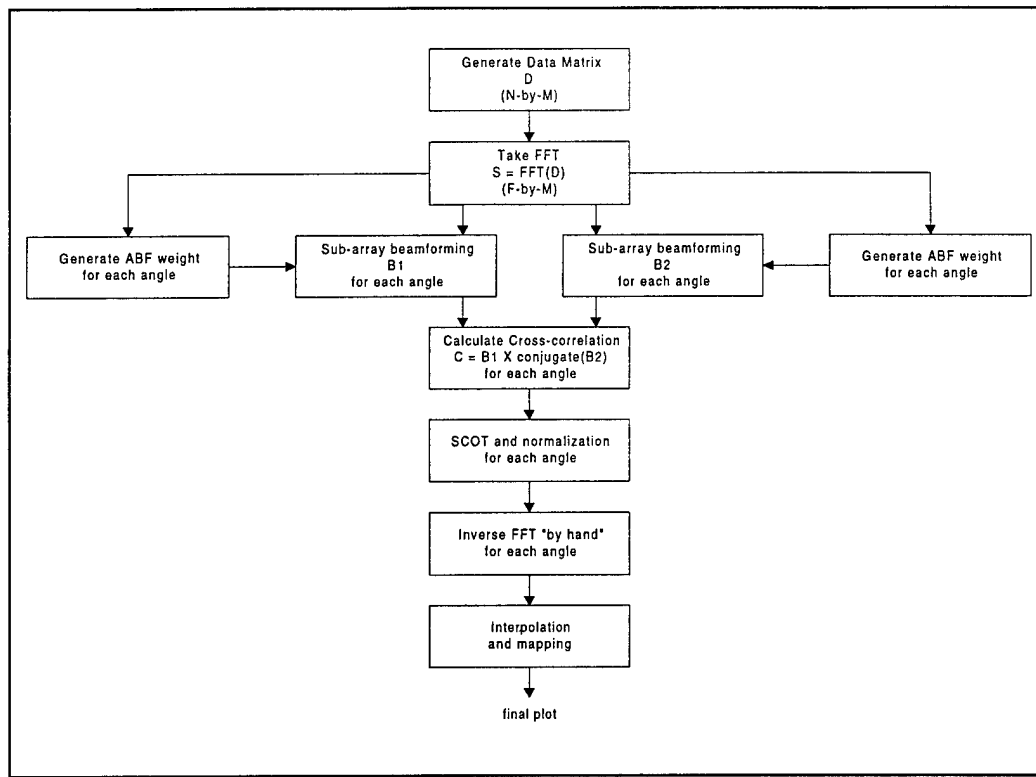


Figure 4.10 – Flowchart for SA-ABF. M : number of nodes, N : number of samples, and F : number of frequency bins.

Whenever we have a new data set, the inverse of the CSM (named matrix R) is calculated for the ABF coefficients. Certain matrix techniques to calculate the inverse matrix are available, such as singular-value decomposition. ABF is only interested in reducing energy from strong interference; therefore, an estimate of R is built using only the first strongest eigenvalues. We add a small amount of white noise identity matrix to make the reduced-dimension matrix full rank. The inverse of the estimated R can be calculated with less computation.

As mentioned earlier, the SA-CBF will perform better than SA-ABF in situations in which nulls are steered toward noise near targets. The ABF weight computation can be modified (in Equation 4.4) to allow control over this situation.

$$w = \frac{(R + \epsilon I)^{-1} v}{v^* (R + \epsilon I)^{-1} v}$$

Equation 4.4

Here, I is the identity matrix and ϵ is a parameter that we are free to choose to match the conditions. We can choose ϵ automatically by finding out the relationship between the white noise gain and ϵ . To show how the ϵ parameter affects the w weights, consider some extreme cases. For $\epsilon = 0$, we have the (strictly) ABF result from above. When ϵ is large, the R matrix is overshadowed by scaled identity matrix, so the weight vector w is equal to the steering vector v . This situation results in SA-CBF.

4.5. Conclusions

The advanced beamforming techniques described in this chapter present new challenges and opportunities for the distributed parallel sonar array. These algorithms provide to the sonar array considerably more computational work than the strictly conventional beamformers. Nonetheless, plans have been laid out for the parallelization of these algorithms in the future. In fact, previous work in parallelizing the single-aperture conventional beamformers will be used extensively in the parallelization of the more advanced versions. This previous work will be taken advantage of by using a modularization approach to the decomposition of the algorithm. With this approach, a fair portion of the parallelization work is already completed. New work will revolve around parallelizing new computations, including the cross-correlation (for the split-aperture algorithm) and the matrix manipulations (in the adaptive algorithm). With the additional capabilities of these algorithms over those of the conventional, single-aperture programs, the DPSA will be able to provide both autonomous performance and high-fidelity beamforming results.

5. Prototype Hardware Architecture

Modeling and simulation have proven to be valuable tools in analyzing and predicting the behavior of computer systems. However, a model can only be as accurate as the assumptions used to construct it. To demonstrate the feasibility of the algorithms being developed, a prototype hardware testbed must be constructed. This testbed should resemble the conceptual architecture as closely as possible while remaining within the time and cost budgets. This section describes the reasoning behind the final hardware selection to be used in construction of the prototype system. Descriptions are given of the processor architecture, the interconnection scheme, and the software development tools that will be used.

5.1. Prototype Hardware Considerations

Several issues play a key role in the development of an efficient and effective sonar array architecture. One of the primary requirements considered for improved mission time and reduced cost is low-power hardware that yields high-performance levels without sacrificing design accuracy and reliability. Advancements in low-power technology, due to increased demand for smaller, faster electronic devices, made the search for prototype hardware challenging. Now that more low-power hardware is available than in the past, selection of high-performance prototype hardware has become a matter of choosing hardware based on other factors in addition to low power consumption. Other technical issues considered in the development of a prototype array include microprocessor speed, memory capacity, communications protocol, and cost. To meet the mathematical processing demands of beamforming algorithms, the chosen architecture must have sufficient processing power and speed. Consequently, digital signal processing (DSP) architectures were considered for use in the prototype hardware.

Digital signal processors (DSPs) are special-purpose processors that differ from general-purpose processors in that they can deal with large amounts of data in real time by repeating simple operations. Real-time performance requires predictable operation, so DSPs have large amounts of on-chip memory with constant access times, as opposed to cache that has unpredictable performance. Most DSPs are equipped with dual-access on-chip memory that allows two operands to be loaded in a single instruction cycle. This feature is critical for faster implementation of filters and other arithmetic operations, which require two memory operands to perform each multiply-accumulate (MAC) operation. Parallelism further enhances this feature through the use of separate program and data memory spaces, thus allowing simultaneous access to program instructions and data and increasing bus bandwidth.

DSP architectures are optimized for computationally-intensive algorithms that often require repetitive FFT calculations, as do many beamforming algorithms. For this reason, they incorporate multiplier units that perform MAC operations in a single instruction cycle. Simple repetitive loops are handled by repeat mechanisms that execute a single instruction or a block of instructions for a predetermined number of times.

Although DSP microprocessors, in the context presented above, may seem to be the most optimal architecture for implementing sonar arrays, general-purpose microprocessors, complete with development boards and starter kits, were also considered for use as node components in the hardware prototype. A feasibility study was conducted to determine suitable architectures that will support the high-performance, fault-tolerant network topology for the prototype. After careful analysis, the *Texas Instruments* C542 DSP chip was the optimal candidate for each node in the sonar array. The C542 is featured on the *Texas Instruments* DSKplus Starter Kit development board (DSKplus). Since the DSKplus has several other features that support the hardware performance characteristics needed by software simulations, the DSKplus was,

ultimately, chosen for each node in the prototype hardware. A myriad of technical support, as well as hardware availability and flexibility, also prompted this choice for the sonar array prototype development.

5.2. Prototype Processor and Basic Node Architecture

The *Texas Instruments* DSP Starter Kit (DSKplus) development board has several features that can address some of the technical challenges of a network-based multicomputer system for large sonar arrays. The DSKplus is a low-cost, high-performance prototyping tool that features the 16-bit fixed-point TMS320C542 DSP chip. The TMS320C54x family of DSPs is so detailed that a separate discussion on its features and characteristics is warranted. A functional overview highlighting some of the practical features of the DSKplus are also discussed, as well as a description of some hardware connections that will be used when implementing the network configuration of the sonar array.

5.2.1. Overview of the Node Processor

There are several characteristics of the *Texas Instruments* TMS320C54x family of DSPs that contribute to high performance levels, as illustrated by the functional diagram in Figure 5.1. Although only the TMS320C542 chip is featured on the DSKplus, it is appropriate to give a general overview of this family of DSPs since the main distinction between individual members of the family lies in on-chip memory organization and peripherals. The *TI* TMS320C54x DSPs (C54s) are 16-bit fixed-point processors that use an advanced derivation of the Harvard-style bus architecture. In conjunction with this standard, there are three separate 16-bit data memory buses and one 16-bit program memory bus in this architecture that contribute to fast data and program memory access times. The program bus (PB) delivers the instruction codes and immediate value operands from the program memory to their appropriate destinations. The three data buses, named CB, DB, and EB, provide connections to on-chip devices such as the CPU, data memory, and the program address generation logic unit (PAGEN). This style of architecture allows for simultaneous access to program instructions and data memory, thus allowing the use of C54s with Dual-Access On-Chip Random Access Memory (DARAM).

The internal memory structure of the C54 is classified into three different memory spaces: program (64K Words), data (64K Words), and I/O (64K Words). C54 devices contain DARAM and/or single-access RAM (SARAM), as well as on-chip read-only memory (ROM). Each C54 device contains different DARAM and SARAM configurations depending on its specific optimizations. It is important to note that the C542 chip has 10K words of on-chip DARAM and 2K words of ROM. The memory structure also includes 26 CPU and peripheral registers that are mapped to the data memory address space.

Another important feature is the CPU, which contains a 40-bit arithmetic logic unit (ALU), a 40-bit barrel shifter, and two 40-bit accumulators named A and B. The CPU also contains a 17 x 17-bit multiplier; a 40-bit adder; and a compare, select, and store unit (CSSU). There are two address generation units included in the CPU: the data address generation unit (DAGEN) which contains several pointers and register, and the PAGEN mentioned previously that houses counters such as the program counter. The functional units in the CPU incorporate a high degree of parallelism, which results in fast execution of mathematical operations. This feature demonstrates the significance of using DSP architectures to exploit single-cycle, computationally-intensive instruction executions that are needed for high-performance implementations of beamforming algorithms. There are numerous other features and on-chip peripherals that enhance the performance of C54s. They include:

- Software-programmable Wait-State Generator and Programmable Bank Switching
- On-chip PLL (Phase Locked Loop) clock generator with internal oscillator or external clock source
- Time-Division Multiplexed (TDM) serial ports that allow simultaneous access to multiple devices
- 8-bit parallel host port interface (HPI)
- Power Consumption Control instructions with Power-Down modes
- 20-ns and 25-ns Single-Cycle Fixed-point instruction execution time for a 3-Volt Power Supply

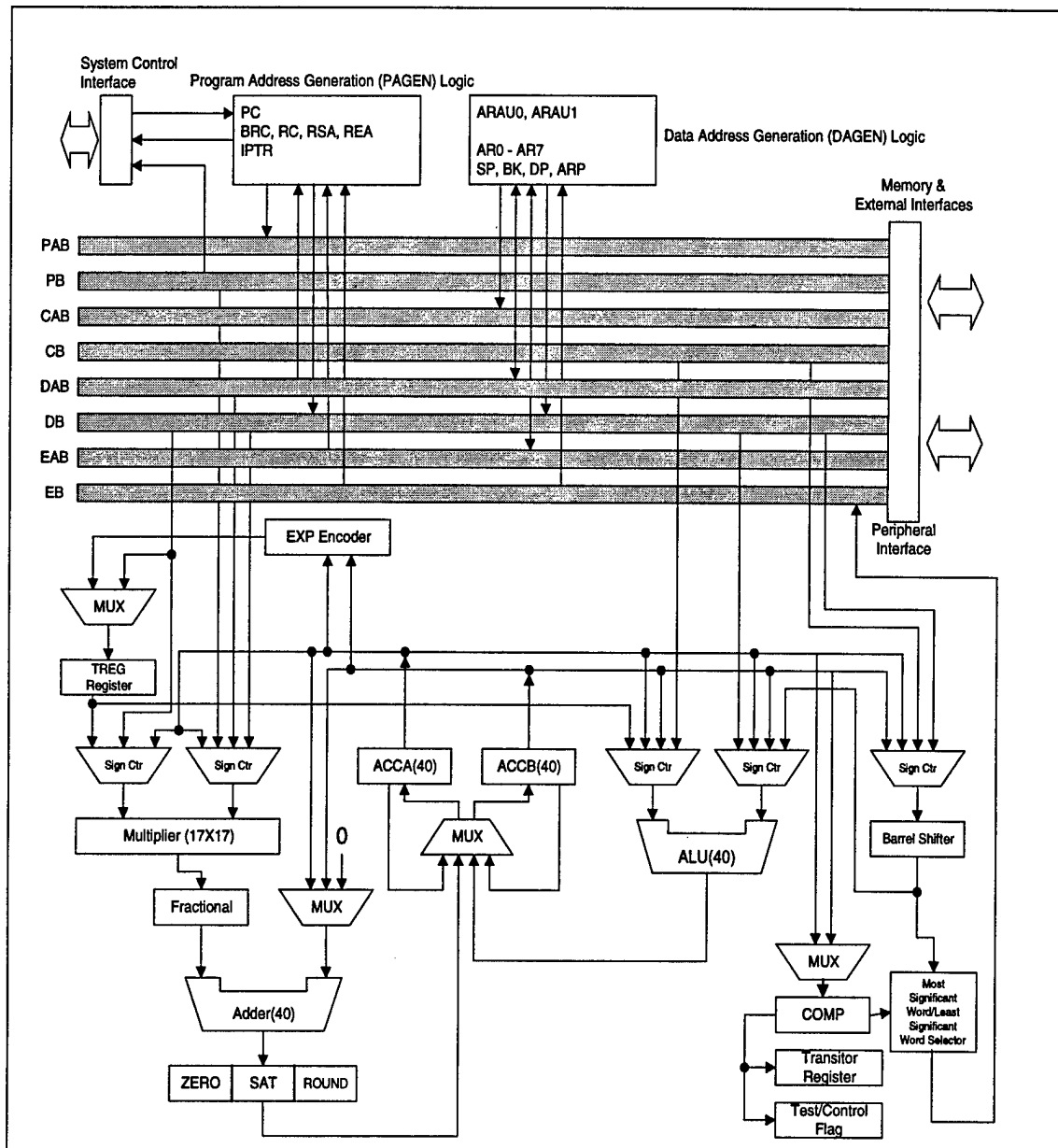


Figure 5.1 - A functional diagram of the TI TMS320C54x Architecture illustrating multiple functional units [TMS97A].

5.2.2. Functional Overview of the Development Board

The numerous features of the C542 chip described establish the desire to use the DSKplus as a basis for the prototype hardware. The DSKplus is a development board that includes other system components that can be used in conjunction with the C542 to produce an optimal prototype architecture. This functional overview of the DSKplus should illustrate some of the reasons why it is one of the most powerful development boards available on the current market, and consequently, a strong contributing factor in its use in the prototype architecture.

As previously mentioned, the DSKplus features the C542 chip that can support the implementation of most real-time DSP algorithms. A board diagram of the DSKplus is shown in Figure 5.2. To adequately meet the algorithmic demands, as well as appropriate hardware and software specifications for conventional and parallel beamforming, the DSKplus also features [TMS96]:

- 14-bit linear resolution programmable ADC/DAC interface circuit for analog-to-digital/digital-to-analog signal conversions,
- buffered serial port (BSP),
- host port-interface (HPI),
- socketed programmable array logic device, PAL22V10, used for reconfiguration of the host port interface for customized designs,
- I/O expansion bus and control signals for external customized designs,
- standard 1/8-inch mono mini-jacks for analog I/O,
- XDS510 emulator header, and
- time-division multiplexed (TDM) serial port header.

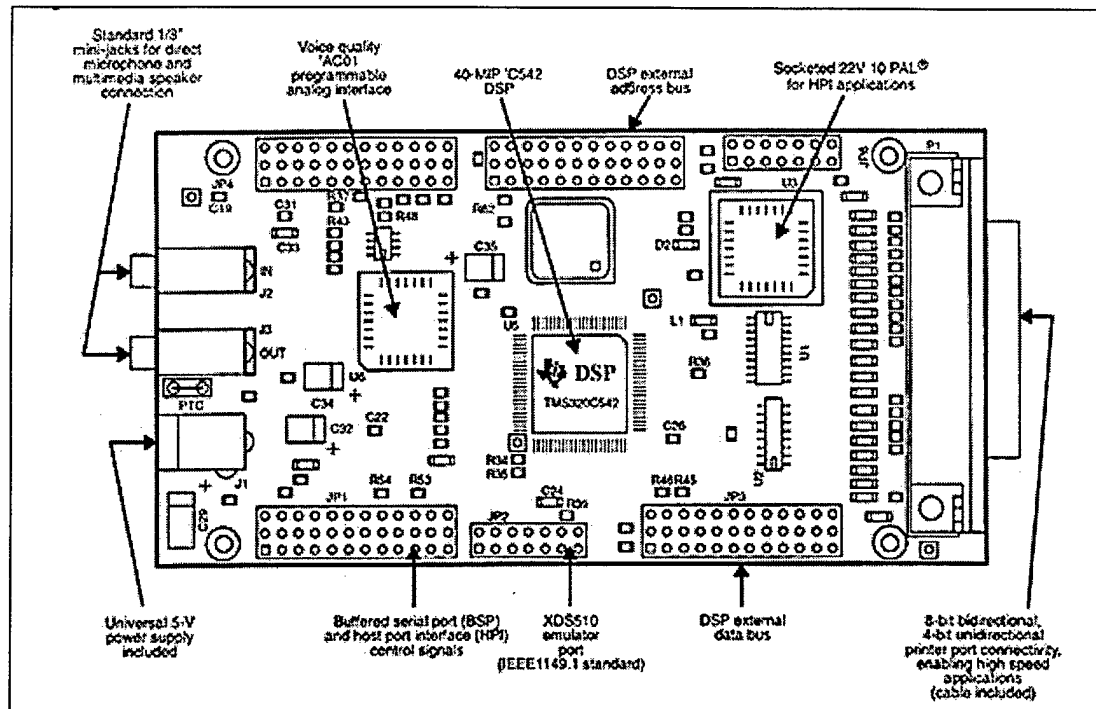


Figure 5.2 – A Board Diagram of DSKplus detailing the components on the board [TMS96].

5.3. Prototype System Configuration

The hardware prototype will consist of eight DSKplus boards physically stacked upon one another, via standoffs. A DB-25 cable is connected between the host PC and one board at a time for downloading the assembled programs. The boards are connected to each other through a daisy-chain configuration where two ribbon cables (one connected to the JP1 TDM header and the other connected to the JP2 emulator header) will be tapped with socket connectors to connect to the male headers on the boards. The male headers are to be constructed and soldered onto the expansion slots on the DSK boards. The ribbon cables will be custom built.

The XDS510 Emulator is an ISA card that is installed into the host PC. The JTAG cable is connected from the XDS510 and buffered before connecting to the first DSK board. Figure 5.3 shows this configuration. The analog interfaces of the DSK boards are not used at this time because the time-series input data will already be written to memory.

The host computer will contain all the software necessary to write, simulate, debug, and test the code for the prototype. This suite includes but is not limited to the optimizing C compiler, Code Explorer, Parallel Debugger Manager, emulation software, etc.

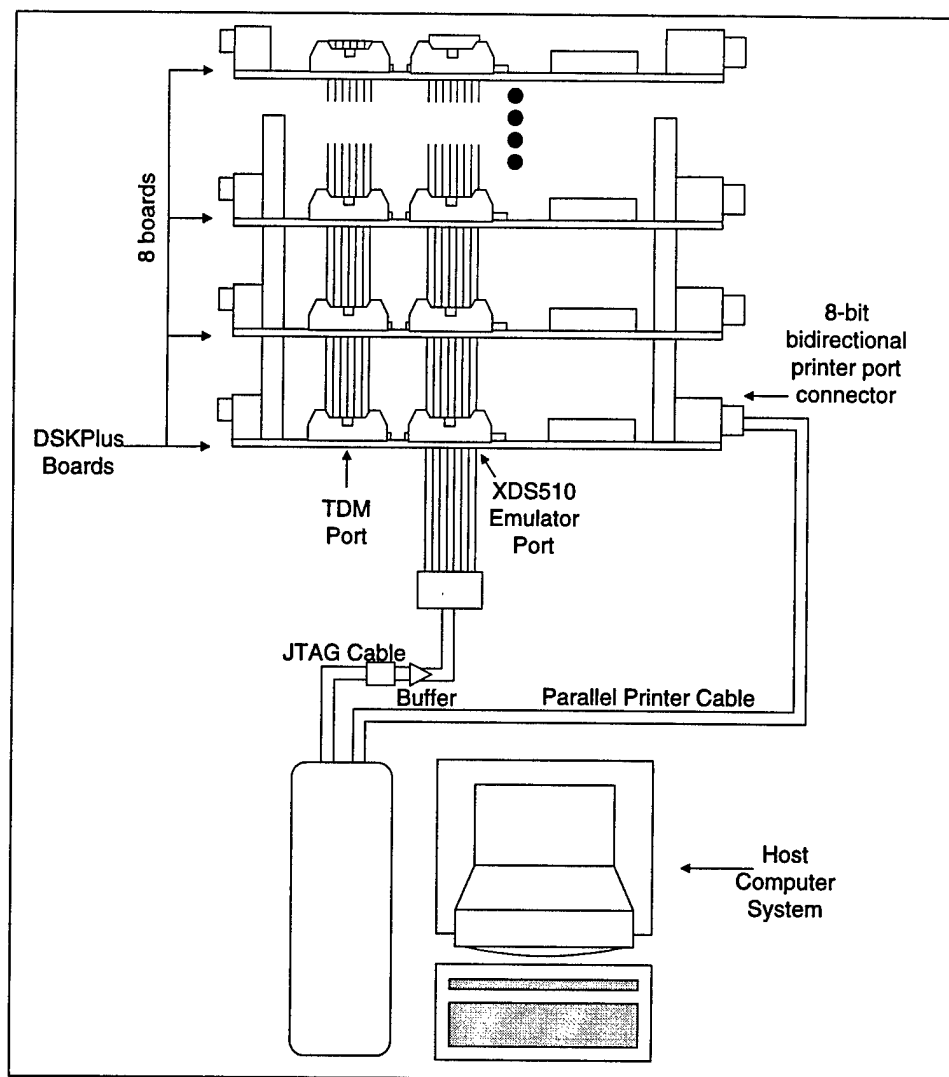


Figure 5.3 – Prototype System Configuration

5.4. Testing and Simulation Environment

Texas Instruments provides a variety of software and hardware tools to code, simulate, and debug DSP applications. Some of these components have already been described as part of the system configuration. The entire suite is described here for completeness. The debugging and simulation tools include the following:

1. DSKplus boards: Hardware interface to the C542 chip. Provides easy communication with host PC and XDS510 Emulator.
2. C Source Debugger: DOS based debugger that allows the user to debug programs in C, assembly, or both.
3. Code Explorer: Windows based simulator that allows the user to view disassembled code in either mnemonic or algebraic instruction set, view memory locations, registers, and a graphical display of data. Communicates with single DSK board through a parallel port. Includes algebraic assembler.

4. XDS510 Emulator: ISA plug-in card that communicates with multiple DSK boards through the JP2 header and JTAG cable. Provides access to on-chip analysis module that lets the user work with *pseudoregisters*, count occurrences of hardware events, and set hardware breakpoints.
5. Parallel Debug Manager (PDM): Interface that allows parallel debugging of code. Multiple debugging screens, one for each processor, are active at the same time.
6. Optimizing C Compiler: Translates ANSI C code to mnemonic assembly code using optimizations specific to C54x devices. Also includes assembler, linker, archiver, and mnemonic-to-algebraic converter.

The assembly language tools work on two different types of instruction sets, the mnemonic instruction set and the algebraic instruction set. The mnemonic instruction set is the conventional type where assembly instructions consist of the instruction mnemonic followed by its operands. The algebraic instruction set is an equation-oriented approach where the operations are written as mathematical equations [TMS97A].

Figure 5.4 below shows the steps taken in developing software for the prototype. The first step is to write the C code for each processor. The code will generally be the same except that the processors will work on different data. The optimizing C compiler will be used to generate the mnemonic assembly code.

The mnemonic assembly source file generated by the compiler is assembled using the mnemonic assembler and converted to algebraic assembly code using the mnemonic-to-algebraic converter. The algebraic assembly code is then assembled using the algebraic assembler. The assemblers generate a Common Object File Format (COFF) file that is used to create an executable by the linker [TMS97C]. The executable is finally loaded to the DSK board through the parallel port.

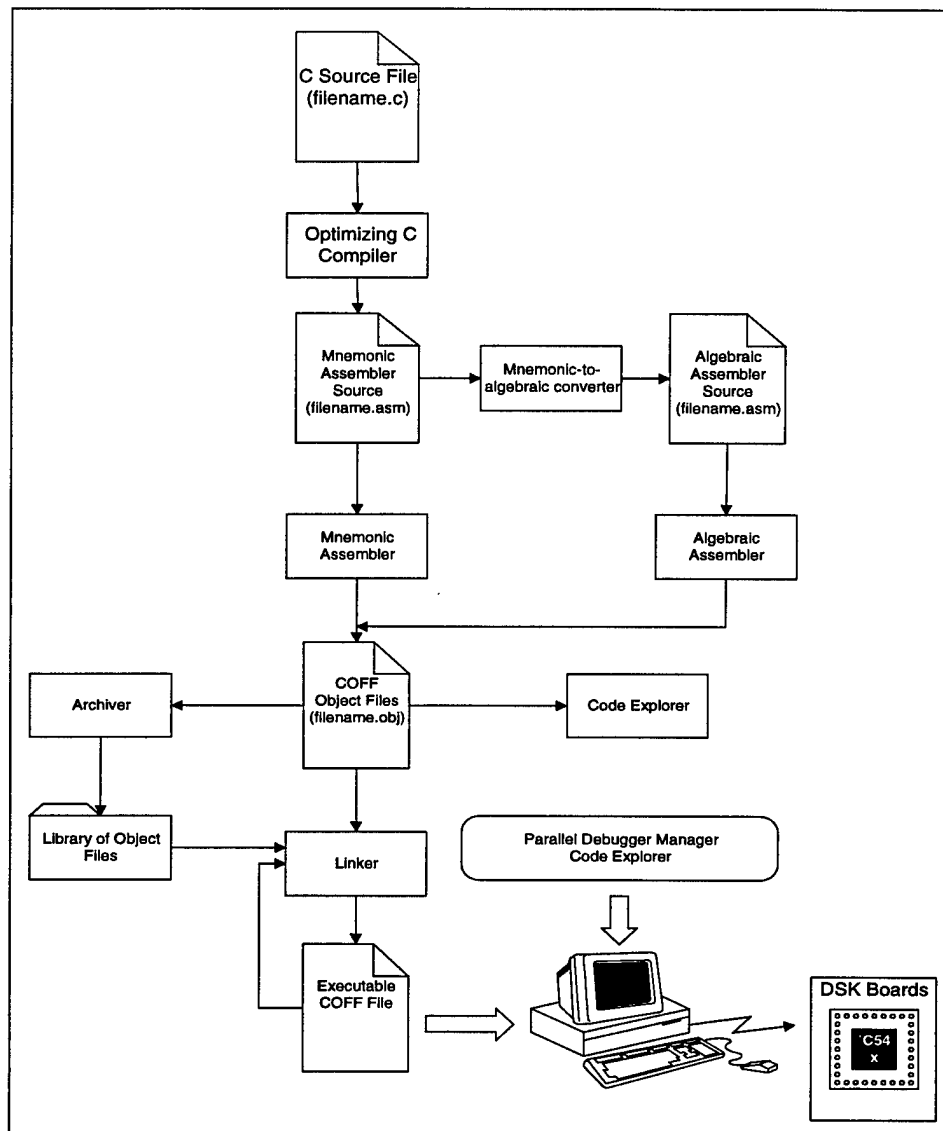


Figure 5.4 - DSP Software Development Steps [TMS97C]

5.4.1. In-circuit Emulator and Debugging Tool

The in-circuit emulator is the XDS510 and is coupled with the parallel debugging manager (PDM). The emulator permits direct control over the processor in the DSK board and gives a good perspective of processor internal operations. It communicates with the DSK boards using a JTAG scanner. The emulator's analysis interface monitors the processor buses and extracts bus cycle information in real time. The emulator can count events such as bus accesses, instruction fetches, CPU clock cycles, and other operations. It can also halt the processor using hardware breakpoints at events such as bus accesses, interrupts, calls/branches taken, and several other events. The debugger provides access to all the processor registers and accumulators, memory, and pipeline *pseudoregisters* that behave as storage registers between pipeline stages [TMS97B].

Before the JTAG cable is connected to the first board, certain signals have to be buffered. These signals are the processor TMS, TDI, TDO, and TCK signals. The buffering helps control the timing skew when driving multiple processors. A resistor value of 4.7k Ω should be used to

pull up the TMS, TDI, and TCK signals to the positive voltage rail in order to hold these signals high when the emulator is not connected. The emulation signals, EMU0 and EMU1, should also be buffered to isolate the processors from the emulator and to properly drive the multiprocessing system [JTAG94].

The PDM allows the user to send commands to multiple processors. Individual processors and groups of processors must be assigned names through a configuration file. The naming of the individual processors associates each individual debugger on the screen with a processor. The debugger provides simultaneous debugging of C and assembly code, interactive data displays, and an easy-to-use interface that facilitates debugging of all eight processors [TMS94].

The debugging environment described above provides all the information needed to easily debug the operations of all eight processors and provides sufficient data for performance analysis. The data being transmitted through the TDM port can be monitored in real time. Throughput and latency measurements can be made through accurate timing analysis.

5.5. Conclusion

The hardware prototype is being developed using the TMS320C542 DSP from *Texas Instruments* as the node processor. The prototype consists of eight processors that will execute the frequency-domain and split-aperture algorithms in parallel while communicating with each other via their TDM ports. This small-scale distributed, parallel array will serve to help analyze issues such as computational speed, communications requirements, robustness, and cost-effectiveness. The choice of the C542 as the prototype node processor was based on practical considerations like low-power operation, weight, size, precision, and computational capability.

System implementation will be simplified by the variety of development tools that *TI* provides. The optimizing C compiler, in-circuit emulator, and parallel debugging manager will be used to develop and debug the system software including the communication implementation.

Conclusions

In the second phase of this project, tasks have extended the study, design, and analysis of the major components in the design and analysis of parallel and distributed computing architectures and algorithms for fault-tolerant sonar arrays. These components include the network architecture, the node processor, and the beamforming algorithm. Furthermore, a robust simulation engine for such systems has been developed. Research has also begun on advanced algorithms for autonomous sonar arrays, including a split-aperture correlational method and adaptive signal processing. The work carried out in this second phase has paved the way for the rapid virtual prototyping of the system and for future developments in parallel beamforming.

The accomplishments for this second phase can be summarized in terms of five research thrusts. First, the components for a laboratory testbed for distributed multicomputer sonar arrays has been acquired and studied. Second, a survey of advanced beamforming algorithms for future use on autonomous sonar arrays has been made. This survey will lead to development of parallel code for such advanced beamformers. Third, the algorithms created during the first year were examined more closely and are now fully understood as far as computation and communication requirements are concerned. Fourth, a robust, fine-grain system simulator based upon the fine-grain network models and parallel algorithms developed in the first phase has been created and exploited. Last, work has begun in studying the fault-tolerant requirements of the DPSA. This work includes hardware bypasses, system software robustness, and beamform algorithm adaptivity.

The prototype development work has led to the acquisition of the hardware components required to build the strawman autonomous sonar array. Extensive study of the internal working of the CPU and the commercial interconnection scheme has been made. These studies will be used in the coming year to create a message-passing software system on the 8-node prototype to run any of the parallel beamformers developed for this project.

The basic structure for the Integrated Simulation Environment was completed during this year. The ISE has provided invaluable results, via its rapid virtual prototyping capability, for use in comparing architectures and algorithms for use on the DPSA. The ISE has provided information on beamformer performance when a wide range of network speeds and processor speeds are used. These parameters can be changed in much the same way (and at the same time) as the granularity knobs in the parallel beamforming algorithms to create a wide range of scenarios. With this information, several conclusions about the capabilities of different versions of the algorithms over different architectural situations were made. A case in point is the pipelined medium-grain beamformer which showed considerable promise on paper, yet did not provide benefits enough (in most cases) to compensate for its additional complexity.

The conventional beamforming research has been completed in this year. The time-domain beamforming algorithms developed last year that lagged in performance were this year augmented. Still, the performance did not surpass that of the FFT beamformers. The important GDS paradigm emerged from the time-domain work. This paradigm provides a level of transparency above the hardware system to the beamform application. In addition to providing an easier interface for the parallel beamformer programmer, the GDS system offers the possibility of many fault-tolerant features. These features range from simple checks for node liveliness to novel concepts in agenda parallelism. The FFT conventional beamforming work was completed with the addition of category breakdowns and with the completion of coding of the algorithms in the taxonomy. Furthermore, an alternative algorithm for the medium-grain program was created to test the possibility of use a high degree of overlap of computation and communication. Results in this research show that the coarse-grain full-capability algorithm slightly outperforms the

medium-grain full-capability algorithms. The medium-grain pipelined alternative shows slight testbed improvement over its non-pipelined cousin. The unidirectional programs show very little speedup over the baseline; thus, there is no reason to run either of the unidirectional algorithms on the sonar array unless a unidirectional network is forced upon us.

Split-aperture and adaptive sonar array beamforming are the algorithms to parallelize for the future. The first work in analyzing and coding a sequential, split-aperture beamformer was completed in this year. In addition, parallelization of the algorithm is underway using knowledge derived from the parallel conventional beamformers. Parallelization of the split-aperture beamformer has the major advantage that the performance of the beamforming output can be improved via the correlation, yet the cost of this improvement in a parallel version is less than the cost in a sequential version. That is, stepping up to the split-aperture algorithm is more easily achieved on autonomous sonar arrays when parallelism is employed.

The last major section of work in this year was the fault-tolerant research into both the architectures and algorithms. The GDS system-level software technology developed in phase two is of particular interest. Not only does it serve as a buffer between the beamforming program and the underlying communications interconnect, but it also provides additional fault-tolerant features. The GDS system provides the beamforming program with local access to the real-time acoustic data so that coding of a data parallel program is much simpler to the traditional sequential programmer. The system provides guaranteed delivery of control messages, is able to track dead nodes and cut them out, and provides synchronization and padding of the several columns of hydrophone data. The system can provide some self-healing to the beamforming programs without the intervention or knowledge of those programs, making the GDS system a robust mechanism for ensuring continued operation of the beamformer in normally damaging situations. All these features will be invaluable to the final development of the DPSSA.

The third phase of this project (which began in January of 1998) involves the development, implementation, and demonstration of a small, laboratory-based hardware prototype with its own distributed, parallel, and embedded software system. This system will be used to verify and validate the simulation results, and in so doing demonstrate and better quantify the inherent advantages of the novel approach employed in its design. Critical factors in the evaluation of the system will center on quantitative measurements in the areas of performance and dependability, including computational speed, efficiency, and precision, reliability, cost, weight, power, size, and mission time, as well as qualitative measurements related to system flexibility, versatility, expandability, scalability, etc.

Bibliography

Prototype Development

- [JTAG94] Texas Instruments. "JTAG/MPSD Emulation Technical Reference." 1994.
- [TMS97A] Texas Instruments. "TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals Guide." 1997.
- [TMS97B] Texas Instruments. "TMS320C54x Optimizing C Compiler User's Guide." 1997.
- [TMS97C] Texas Instruments. "TMS320C54x Assembly Language Tools User's Guide." 1997.
- [TMS96] Texas Instruments. "TMS320C54x DSKplus DSP Starter Kit User's Guide." 1996.
- [TMS94] Texas Instruments. "TMS320C5xx C Source Debugger User's Guide." 1994.

Beamforming, General

- [BIEN77] Bienvenu, G. "An Adaptive Approach to Underwater Passive Detection." *Aspects of Signal Processing, Part 1*. Ed. G. Tacconi. Dordrecht, Holland: D. Reidel, 1977: 395-400.
- [BURD91] Burdic, William S. *Underwater Acoustic System Analysis*. 2nd ed. New Jersey: Prentice Hall, 1991.
- [CAND94] Candy, J. V. and E. J. Sullivan. "Model-Based Processing for a Large Aperture Array." *IEEE Transactions on Oceanic Engineering* 19.4 (Oct. 1994): 519-28.
- [CHEN82] Chen, C. H. *Digital Waveform Processing and Recognition*. Boca Raton: CRC Press, 1982.
- [CHEN88] Chen, C. H. ed. *Signal Processing Handbook*. New York: Marcel Dekker, 1988.
- [CHIN94] Ching, P. C. and H. C. So. "Two Adaptive Algorithms for Multipath Time Delay Estimation." *IEEE Transactions on Oceanic Engineering* 19.3 (Jul. 1994): 458-62.

- [CROC81] Crochiere, Ronald E. and Lawrence R. Rabiner. "Interpolation and Decimation of Digital Signals - A Tutorial Review." *Proceedings of the IEEE* 69.3 (Mar. 1981): 300-31.

- [DEFA88] DeFatta, David J., Joseph G. Lucas, and William S. Hodgkiss. *Digital Signal Processing: A System Design Approach*. New York: Wiley, 1988.

- [ERMO94] Ermolaev, Victor T. and Alex B. Gershman. "Fast Algorithm for Minimum-Norm Direction-of-Arrival Estimation." *IEEE Transactions on Signal Processing* 42.9 (Sep. 1994): 2389-93.

- [FARR77] Farrier, D. R. and T. S. Durrani. "Signal Extraction Algorithms for Adaptive Processing of Array Data." *Aspects of Signal Processing Part 2*. Ed. G. Tacconi. Dordrecht, Holland: D. Reidel, 1977: 485-94.

- [FUCH95] Fuchs, Jean-Jacques. "Shape Calibration for a Nominally Linear Equispaced Array." *IEEE Transactions on Signal Processing* 43.10 (Oct. 1995): 2241-8.

- [GERS95] Gershman, Alex B., Victor I. Turchin, and Vitaly A. Zverev. "Experimental Results of Localization of Moving Underwater Signal by Adaptive Beamforming." *IEEE Transactions on Signal Processing* 43.10, (Oct. 1995): 2249-57.

- [GOODa] Goodwin, Michael, and Gary Elko. "Constant Beamwidth Beamforming Using an Affine Phase Multi-Beamformer." Murray Hill, New Jersey: AT&T Bell Laboratories, Acoustics Research Department.
<http://ptolemy.eecs.berkeley.edu/~michaelg/constant.ps> .

- [GOODb] Goodwin, Michael. "Frequency-Independent Beamforming." Berkeley, CA: U of Berkeley, EECS, and AT&T Bell Laboratories, Acoustics Research.
<http://ptolemy.eecs.berkeley.edu/~michaelg/fib.ps> .

- [GORI] Goris, Malcolm J., and Donald J. Mclean. "Towed Array-Shape Estimation: A Comparison of Methods." CSIRO, Division of Radiophysics.
<http://www.rp.csiro.au/people/mgoris/papers/asecomp.ps> .

- [GRIF77] Griffiths, J. W. R., and J. E. Hudson. "An Introduction to Adaptive Processing in a Passive Sonar System." *Aspects of Signal Processing, Part 1*. Ed. G. Tacconi. Dordrecht, Holland: D. Reidel, 1977: 299-308.

- [GRIF73] Griffiths, J. W. R., P. L. Stoklin, and C. van Schooneveld, eds. *Signal Processing; Proceedings*. Academic Press: London, 1973.

- [HAMP93] Hampson, Grant and Andrew P. Papliński. "Beamforming by Interpolation." Technical Report 93/12. Clayton, Victoria, Australia: Monash U, 1993.
ftp://ftp.rdt.monash.edu.au/pub/techreports/RDT/93-12.ps.Z .

- [HAMP95] Hampson, Grant and Andrew P. Papliński. "Simulation of Beamforming Techniques for the Linear Array of Transducers." Technical Report 95/3. Clayton, Victoria, Australia: Monash U, 1995. ftp://ftp.rdt.monash.edu.au/pub/techreports/RDT/95-3.ps.Z .

- [HAYK91] Haykin, Simon. *Adaptive Filter Theory*. 2nd ed. Englewood Cliffs: Prentice Hall, 1991.

- [HAYK95] Haykin, Simon, ed. *Advances in Spectrum Analysis and Array Processing Volume III*. Englewood Cliffs: Prentice Hall, 1995.

- [HOLM] Holm, Sverre. "Digital Beamforming in Ultrasound Imaging." Oslo, Norway: U of Oslo, Department of Informatics. ftp://ftp.ifi.uio.no/pub/publications/others/SHolm-4.ps.Z .

- [HORV92] Horvat, Dion C. M., John S. Bird, and Martie M. Goulding. "True Time-Delay Bandpass Beamforming." *IEEE Transactions Oceanic Engineering* 17.2 (Apr. 1992): 185-92.

- [IFEA93] Ifeachor, Emmanuel C. *Digital Signal Processing: A Practical Approach*. Workingham, England: Addison-Wesley, 1993.

- [JOHN93] Johnson, Don H. and Dan E. Dudgeon. *Array Signal Processing, Concepts and Techniques*. New Jersey: Prentice Hall, 1993.

- [JOY97] Joy, Kenneth I. "Bresenham's Algorithm." Davis, CA: Visualization and Graphics Research Laboratory, U. of California, Davis, 1997.

- [LI95] Li, Shaolin and Terrence J. Sejnowski. "Adaptive Separation of Mixed Broad-Band Sound Sources with Delays by a Beamforming Héault-Jutten Network." *IEEE Transactions on Oceanic Engineering* 20.1 (Jan. 1995): 73-8.

- [LUND77] Lunde, E. B. "The Forgotten Algorithm in Adaptive Beamforming." *Aspects of Signal Processing, Part 2*. Ed. G. Tacconi. Dordrecht, Holland: D. Reidel, 1977: 411-21.

- [MCWH89] McWhirter, J.G and T.J. Shepherd. "Systolic Array Processor for MVDR Beamforming," *IEE Proceedings* 136.F2 (Apr. 1989):75-80.

- [MORG94] Morgan, Don. *Practical DSP Modeling, Techniques, and Programming in C*. New York: Wiley, 1994.

- [NIEL91] Nielsen, Richard O. *Sonar Signal Processing*. Boston: Artech House, 1991.

- [NORD94] Nordebo, Sven, Ingvar Claesson, and Sven Nordholm. "Adaptive Beamforming: Spatial Filter Designed Blocking Matrix." *IEEE Transactions on Oceanic Engineering* 19.4 (Oct. 1994): 583-9.

- [PILL89] Pillai, Unnikrishna S. *Array Signal Processing*. New York: Springer-Verlag., 1989.

- [SMIT95] Smith, Winthrop W. and Joanne M. Smith. *Handbook of Real-Time Fast Fourier Transforms: Algorithms to Product Testing*. New York: IEEE, 1995.

- [TANG94] Tang, C. F. T., K. J. R. Liu, and S. A. Tretter. "Optimal Weight Extraction for Adaptive Beamforming Using Systolic Arrays," *IEEE Transactions on Aerospace and Electronics Systems* 30.2 (Apr. 1994): 367-84.

- [TRAN93] Tran, Jean-Marie Q. D. and William S. Hodgkiss. "Spatial Smoothing and Minimum Variance Beamforming on Data from Large Aperture Vertical Line Arrays." *IEEE Transactions Oceanic Engineering* 18.1 (Jan. 1993): 15-23.

- [VANP95] Vanpoucke, Filiep and Marc Moonen. "Systolic Robust Adaptive Beamforming with an Adjustable Constraint," *IEEE Transactions on Aerospace and Electronic Systems* 31.2 (Apr. 1995): 658-68.

- [VIBE95] Viberg, Mats, Björn Ottersten, and Arye Nehorai. "Performance Analysis of Direction Finding with Large Arrays and Finite Data." *IEEE Transactions on Signal Processing* 43.2 (Feb. 1995): 469-76.

- [YU95] Yu, Jung-Lang and Chien-Chung Yeh. "Generalized Eigenspace-Based Beamformers." *IEEE Transactions on Signal Processing* 43.11 (Nov. 1995): 2453-61.

- [ZVAR93] Zvara, George P. "Real Time Time-Frequency Active Sonar Processing: A SIMD Approach," *IEEE Journal of Oceanic Engineering* 18.4 (Oct. 1993): 520-8.

Networks

- [BERT92] Bertsekas, Dimitri and Robert Gallager. *Data Networks*. New Jersey: Prentice Hall, 1992.
- [FISH95] Fishwick, Paul A. *Simulation Model Design and Execution, Building Digital Worlds*. New Jersey: Prentice Hall, 1995.
- [HAMM88] Hammond, Joseph L. and Peter J. P. O'Reilly. *Performance Analysis of Local Computer Networks*. Massachusetts: Addison Wesley, 1988.
- [SPUR95] Spurgeon, Charles. *Guide to 10-Mbps Ethernet*. Austin, Texas: U. of Texas, Austin Networking Services, 1995.
- [STAL94] Stallings, William. *Data and Computer Communications*. New York: Macmillan, 1994.

Parallel Computing

- [AMDA67] Amdahl, G. M. "Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities." *Proc. AFIPS* 30 (1967): 483-5.
- [BAL90] Bal, H.E. *Programming Distributed Systems*. New Jersey: Silicon Press, 1990.
- [BECK96] Beck, Alan, ed. "Dave Gustavson Answers Questions About SCI," *HPCWire*, Oct. 4 1996.
- [CARR90] Carriero, Nicholas and David Gelernter. *How to Write Parallel Programs: A First Course*. Cambridge: MIT Press, 1990.
- [CORM95] Cormen, Thomas H. and Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithms*. New York: McGraw-Hill, 1995.
- [FLYN72] Flynn, M.J. "Some Computer Organizations and Their Effectiveness." *IEEE Trans. Computers* 21.9 (1972): 948-60.

- [FORE96] *ForeRunner™ SBA-200 ATM Sbus Adapter User's Manual, MANU0069, Version 4.0*, FORE Systems, Inc., Rev. A-March 1996.
- [FORT78] Fortune, S. and J. Willie. "Parallelism in Random Access Machines." *Proc. 10th Annual ACM Symp. on Theory of Computing* (1978): 114-8.
- [FOST95] Foster, Ian. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Reading, Mass: Addison-Wesley, 1995.
- [GIBB88] Gibbons, Alan and Wojciech Rytter. *Efficient Parallel Algorithms*. Cambridge: Cambridge U. Press, 1988.
- [GROP] Gropp, William, Ewing Lusk, Nathan Doss, and Anthony Skjellum. "A High-performance, Portable Implementation of the MPI Message-Passing Interface Standard."
- [GROP95] Gropp, William and Ewing Lusk and Anthony Skjellum. *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. Massachusetts: MIT Press, 1995.
- [GROP96] Gropp, William and Ewing Lusk. *User's Guide for mpich, a Portable Implementation of MPI*, Chicago: U. of Chicago, 1996.
- [GUST88] Gustafson, J. L. "Reevaluating Amdahl's Law." *Commun. ACM* 31.5 (May 1998): 532-3.
- [HERR91] Herrarte, Virginia and Ewing Lusk. "Studying Parallel Program Behavior with *Upshot*." Technical Report ANL--91/15. Argonne, IL: Argonne National Laboratory, 1991.
- [HWAN93] Hwang, Kai. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. New York: McGraw-Hill, 1993.
- [LADD95] Ladd, Scott Robert. *C++ Templates and Tools*. New York: M&T Books, 1995.
- [LEWI92] Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing*. New Jersey: Prentice-Hall, 1992.

- [LIN83] Lin, Shu and Daniel J. Costello, Jr. *Error Control Coding: Fundamentals and Applications*. New Jersey: Prentice Hall, 1983.
- [MESS94] Message-Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Knoxville: U. of Tennessee, May 1994.
- [NICH96] Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. Cambridge: O'Reilly, 1996.
- [PAIG93] Paige, Robert, John Reif, and Ralph Wachter, eds. *Parallel Algorithm Derivation and Program Transformation*. Boston: Kluwer, 1993.
- [PROG95] "Programmer's Guide to MPI for Dolphin's Sbus-to-SCI Adapters Version 1.0." U. of Bergen: Parallab, Nov 1995.
- [QUIN94] Quinn, Michael J. *Parallel Computing: Theory and Practice*. New York: McGraw-Hill, 1994.
- [RAGS91] Ragsdale, Susann, ed. *Parallel Programming*. New York: McGraw-Hill, 1991.
- [SNIR96] Snir, Marc, S. et. al. *MPI: The Complete Reference*. Cambridge: MIT Press, 1996.
- [TAKE92] Takeuchi, Akikazu. *Parallel Logic Programming*. New York: Wiley, 1992.
- [ZOMA96] Zomaya, Albert, ed. *Parallel and Distributed Computing Handbook*. New York: McGraw-Hill, 1996.

UNIX Programming

- [ROBB96] Robbins, Kay A. and Steven Robbins. *Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading*. Upper Saddle River, N.J: Prentice Hall, 1996.
- [STEV92] Stevens, W. Richard. *Advanced Programming in the UNIX Environment*. Massachusetts: Addison Wesley, 1992.

Appendix A. Extensions to Integrated Simulation Environment

This appendix contains documentation regarding the use and details of ISE. It is intended to provide extra information beyond the high-level exposure provided in the main body of the text.

A.1. Integrated Simulation Environment Internals

The Integrated Simulation Environment is composed of a number of components, as was shown in Figure 2.1, which work together during a simulation. First, the user's parallel program is comprised of several processes, each of which is an instance of the MPI executable. In the ISE, all of the parallel processes for a given iteration run on the same machine, called the local machine or the process machine. Second, the shared files in the system are the means by which the different entities communicate. Third, the relay programs serve to move information between the shared file on the local machine and the shared file on the machine running the BONEs simulation, called the remote machine or the BONEs machine. When the parallel processes are sending information to BONEs, the local relay process (the leftmost relay in Figure 2.1) multiplexes the data from the different columns of the shared file into a single socket. The remote relay demultiplexes the data to place it in the correct column of the remote shared file. The opposite situation occurs when the BONEs simulation sends data to the parallel processes. Fourth, the BONEs simulation for a particular iteration is running on the remote machine.

The various ISE entities are all spawned by a single program called *ise*. The *ise* program's responsibility is to spawn a *bmpi* instance for each iteration. Each *bmpi* program spawns the parallel processes and the local relay process for its iteration. It also creates the shared files on the local machine and initializes their values. When the BONEs simulation is started, a primitive in the Portal block creates the remote shared files and spawns the remote relay.

A.1.1. Communication Structures

There are three different types of shared-memory files that are used in the ISE to enable the communication between the parallel MPI processes and the BONEs simulation. First, the bridge file provides information that is static to a particular iteration of the simulation. The structure name arises from the fact that such static information can be placed in a single file, yet be read by processes on the two machines. Thus, the bridge file bridges the gap between the BONEs simulation on the remote machine and the MPI processes on the local machine. Second, the share file, which has an incarnation on both the remote machine and the local machine, contains information that is shared between the processes on the machine. Third, a BONEs file for each machine contains columns for each rank of the parallel program and is the major component of the communication between a rank of the MPI processes and a rank of the BONEs simulation.

The structure of the bridge file is shown in Figure A.1 below. The file contains a floating-point value for the version of the ISE currently running, an integer for number of iterations in the simulation, an integer for number of nodes per iteration, a string for the name of the local machine, an integer for the socket port number, and an integer flag for silent mode. There is a bridge file for each iteration. Bridge files are named ".bmpi_bridgefilexx," where the "xx" is the iteration number to which the bridge file belongs.

ISE Version
Number of Iterations
Number of Nodes
Local Machine Name
Relay Port
Silent Mode Flag

Figure A.1 – Structure of the ISE Bridge File

The ISE version field is used by the various entities in the simulation to make sure they are all using the same version of the ISE. If any of the entities realizes that its version number does not match with the version number it finds in the bridge file for its iteration, it prints an error message to the user and terminates. Entities with different version numbers cannot work together in most cases because the shared-memory files and mutual exclusion procedures may be different. These problems may cause a range of errors including segmentation faults, inability to find communication files, and incorrect timing of code blocks.

The field for the number of iterations indicates the number of iterations that have been spawned. This number is set by the user during the execution of the *ise* runtime program. The definition of “iteration” used here is a BONEs iteration. BONEs is capable of concurrently running the same simulation with different parameters. The *ise* program must be told the number of iterations so that it can spawn the correct number of MPI parallel systems to match the simulations BONEs will spawn.

During startup, the user also specifies the field for number of nodes. The number of nodes is used in several places to define the size of the iteration. Such places include calls to *MPI_Comm_size* and creation of the correct number of columns in the BONEs files. One limitation that the ISE places on the user is that all iterations that are running at the same time must have the same number of nodes. This limitation is acceptable because there exist very few situations in which BONEs will be able to spawn multiple iterations with different numbers of nodes per iteration.

The fields for local machine and relay port are used by the remote relay program to find the local machine. Because BONEs is unpredictable in relation to which machines (of the list provided by the user) it will put particular iterations, the ISE must provide a method for matching the remote machine to the local machine of the correct iteration number. These fields are designed to allow the remote relay to search out the correct local relay.

The silent mode flag is simply an indication to the remote machine of whether or not the local machine is in a debugging mode. If the local machine is in a debugging mode, the silent mode will be turned off, and the BONEs simulation will spawn a window for the remote relay, which will then print out debugging information. If the local machine is in silent mode, the remote relay will run in the background without a dedicated window and will not print anything to the user.

The format of the share file is shown in Figure A.2. For each iteration, there is a share file on both the local machine with the MPI processes and the remote machine with the BONEs simulation. The reason that there must be a share file on each machine is that the file contains fields that dynamically change during the simulation. Such dynamic information does not

propagate correctly across two separate machines; therefore, processes can only use this file for communication with other processes on the same machine.

Rank
Time Factor
Spawn-Finish Semaphore
Simulation-Finish Semaphore
Parameters Bit Field
Signal-To-BONeS Mutex
Signal-To-Process Mutex
Full-Notime-Slot Mutex
Empty-Notime-Slot Mutex
BONeS Machine Name
Remote Relay PID
Execution Mutex

Figure A.2 – Structure of the ISE Share File

The rank field is an integer that aids in the spawning of the MPI processes. The MPI processes are created one at a time. When the process starts, it will read its rank from this field in the local share file. Once the rank is safely placed in a variable within the process's address space, the process will signal the spawning program, *bmpi*, that it is safe. The *bmpi* program can then overwrite this rank field with a new rank and spawn a new MPI process.

The time factor field is the floating-point number that specifies the performance scaling chosen by the user for the CPUs in this iteration. This performance scaling was described above. Note that the time factor is uniform for all MPI processes in an iteration but may be different for different iterations.

The spawn-finish semaphore and simulation-finish semaphore are used for signaling the spawning program of when the spawn has started safely and when the spawned program is finished, respectively. A semaphore is a computing structure that allows two basic functions. Posting to a semaphore means that the value of the semaphore counting variable is incremented. The semaphore ensures that this value is incremented atomically; that is, no other process is able to modify the value while one process is posting. Waiting on a semaphore means that the counting variable is atomically decremented. If the value is zero, the semaphore wait causes the process to block until another process posts to the semaphore, thus increasing the counting value above zero. In the ISE, semaphores are used extensively as signals between different entities. The spawn-finish semaphore is initially set to 0. After spawning an MPI process, the spawning program, *bmpi*, waits on the spawn-finish semaphore. Since this semaphore has value 0, the *bmpi* program is blocked. Once the MPI process has opened the local share file and read its rank successfully, it posts the spawn-finish semaphore. The *bmpi* program will then become unblocked and continue spawning more MPI processes. The same signaling procedure is used

with the simulation-finish semaphore. After spawning every MPI process for the iteration, *bmpi* will wait on the simulation-finish semaphore. When the MPI processes are complete, they post to this semaphore so that *bmpi* can return from the wait and terminate successfully.

The bit field for parameters contains information for the MPI processes on whether or not to use jump-start and profiling. If the profiling bit in this field is set, then the MPI processes for this iteration will record the times of communications and the nodes involved into a profiling file. This file can then be used after the simulation to view with the Upshot graphical profiling tool. The jump-start bit indicates to the MPI processes whether or not they should reset their internal timing clocks to zero at the first MPI call. This procedure effectively means that all the code before the first MPI call in an MPI process executed in zero time. The jump-start feature may be used to skip simulation of any setup computations the nodes perform before getting into the desired work.

The signal-to-BONeS mutex and signal-to-process mutex are part of a system of communication between the BONeS simulation and the MPI processes. The method is that of signaling with semaphores. When an MPI process wants to wait for a signal from its BONeS counterpart, it waits on a signal-to-process semaphore. To signal the process, the BONeS simulation will post this semaphore. A problem arises from the fact that one may want to simulate many nodes but the operating system puts a limit on the number of semaphores that can be used by a process. With 32 processes in a simulation, the relay programs would need access to 32 semaphores for BONeS-to-process communication and 32 semaphores for process-to-BONeS communication, putting the number of semaphores over the imposed limit. To bypass this limit, the ISE uses what are here called *pseudo-semaphores*. A single pseudo-semaphore is composed of a mutex (one of either the signal-to-BONeS mutex or the signal-to-process mutex) and an integer (which is stored in the BONeS file, discussed next). In the situation of the above example, there are 32 integers for BONeS-to-process communication and 32 integers for process-to-BONeS communication. There is only one mutex each for BONeS-to-process communication and process-to-BONeS communication. When a program wants to signal a process, it locks the signal-to-process mutex in the share file to ensure atomicity, increments the signal-to-process integer for the correct process in the BONeS file, and unlocks the mutex. The situation is similar for signaling to BONeS. To wait on a signal, a program will lock the mutex and check the integer value. If the value is greater than 0, it decrements the value, unlocks the mutex, and returns. If the value is 0, it unlocks the mutex, waits a short while, and tries again, thus giving some other program the opportunity to post the pseudo-semaphore in the meantime. Using these pseudo-semaphores, any limitation on the number of real semaphores or mutexes is avoided because there are only two mutexes for all the pseudo-semaphores. The signal-to-BONeS pseudo-semaphore is used by one of the MPI processes to signal its counterpart node in the BONeS simulation that it has a network request to make or it has some information to send. The signal-to-process pseudo-semaphore is used by a BONeS node to signal its counterpart MPI process with an acknowledgement or with the requested data. The next fields in the share file, the *notime* mutexes, are part of pseudo-semaphores used for *notime*, direct communication between the MPI processes without network interaction.

The BONeS machine name and remote relay process ID fields are used for unusual termination of the simulation. At startup, when the socket between the remote relay and the local relay is established, the local relay is ignorant of how the connection was made. Recall, it is the remote relay that seeks out the local relay. The first information communicated over the socket is the name of the remote machine and the process ID of the remote relay. The local relay puts this information in the local share file. If the user terminates the simulation prematurely, the *bmpi* program will use this information to spawn to the remote machine and kill the remote relay. The actual BONeS simulation on the remote machine must be killed manually by the user.

The last field in the share file is the execution mutex. This mutex is used on the local machine only and is shared by all the entities on that machine. The goal of the execution mutex is to provide the MPI processes with exclusive access to the CPU so that they can get their code block timings without being interrupted by other ISE components. If other entities are allowed to execute while an MPI process is timing a code block (this includes the other MPI processes), then the code block timings will be falsely augmented. To correct this problem, each entity on the local machine must lock the execution mutex before doing anything and release it before waiting on a semaphore or a pseudo-semaphore. Using this procedure, the MPI processes are ensured that when they lock the mutex and begin to time a code block, no other entity within the ISE can swap it out and mar the code block timings. Other processes not affiliated with the ISE have no knowledge of this execution mutex and do not lock it when executing; therefore, the machine on which the MPI processes are running should be free of activity from other programs or other users in order to get code block timings of an unloaded machine.

The BONEs file structure is shown in Figure A.3. There are two BONEs files, a local copy and a remote copy, for each iteration. The purpose of the relay programs is to relay the dynamic information in the BONEs file on one side of the socket on one machine to the BONEs file on the other machine. Each BONEs file has a column for each node. The communication between an MPI process and its counterpart node in the network simulation is done completely through this column in the BONEs files.

<i>Node 0 Column</i>	<i>Node 1 Column</i>	<i>Node 2 Column</i>	<i>Node 3 Column</i>
Request (which call and at what time)	Request (which call and at what time)	Request (which call and at what time)	Request (which call and at what time)
Packet Header (source, destination, tag, size)	Packet Header (source, destination, tag, size)	Packet Header (source, destination, tag, size)	Packet Header (source, destination, tag, size)
Signal-to-Process Int	Signal-to-Process Int	Signal-to-Process Int	Signal-to-Process Int
Signal-to-BONEs Int	Signal-to-BONEs Int	Signal-to-BONEs Int	Signal-to-BONEs Int
Signal Type	Signal Type	Signal Type	Signal Type
Data Segment	Data Segment	Data Segment	Data Segment

Figure A.3 – Structure of the ISE BONEs File

The request field contains a structure for making network communication requests and returning network time information. For processes making requests to BONEs, the request field contains the type of MPI call requested and the time at which it is to be called. Each process times its code block, reports to BONEs the time of the MPI call and which call it is, and waits for BONEs to reach that point in time and initiate the call. When the BONEs simulation is finished with an MPI call, it reports the time of completion to the MPI process through the time field in this structure.

The packet header field contains information necessary to complete MPI calls. For example, an *MPI_Recv* requires the receiver to specify the message envelope (source from which to receive and tag to receive) to the MPI implementation. This information is located in this packet header field when BONEs initiates the *MPI_Recv*. This field is also used by BONEs to inform the MPI process of such information as the exact message envelope received (in case the MPI process had specified *MPI_ANY_SOURCE* or *MPI_ANY_TAG*) or the actual data size received.

The signal-to-process integer and signal-to-BONEs integer in each column are part of the array of pseudo-semaphores discussed above.

The MPI processes and the BONEs simulation use the signal type field to inform the relay programs about the type of information being sent. The relay programs use a fixed-length message format when communicating with each other over the socket. This message structure includes a data size field but does not include room for any data. If the MPI processes and BONEs simulation want to communicate some data, they inform the relay that the signal type is "send data." When the relay creates the fixed-size message to send across the socket, this signal type is included so that the relay on the opposite end of the socket will know there is additional information to be retrieved. The originating relay will then send the variable-sized data through the socket. The opposite relay will read the correct amount thanks to the signal type field and the data size field in the packet header. If there is no data to send, the signal type is set to "send no data." If the message is an acknowledgement from BONEs to a request made by a process, the signal type is set to "acknowledge request." This setting specifies to the relay programs that they should check to see if the request was an *MPI_Finalize*. If so, then the relays decrement their count of how many MPI processes remain alive. When all MPI processes have received acknowledgements to *MPI_Finalize*, the relay programs can terminate.

The last field in the BONEs file is the data field. The data field is used when an MPI process wants to send data through the network or when a BONEs node receives a message over the network and returns it to the MPI process. Even though there is no limit on the size of the data that can be sent with an MPI call, the data field in this file is a fixed-length field. When a process wants to send more data than there is room in the field, it initiates a piecewise transfer operation with the local relay. The process will place the first part of the data in the field and wait for a handshake from the local relay (via the pseudo-semaphore in the column for that process). When the handshake arrives, the process knows that the local relay has copied that portion of the data into its memory space so that it is safe to overwrite the data field in the BONEs file with the next piece of the large data message.

A.1.2. Process Structure and Spawning

The MPI processes are created by the user in the same way the user would create any other MPI parallel program. The user includes the *MPI_Init* command at the beginning of the MPI program, calls the standard MPI functions within the program, and ends with the *MPI_Finalize* command. The user includes the *mpi.h* header library file and compiles with the *libmpi.a* object archive file. The user must simply use the correct path for the ISE versions of these files instead of the versions used by any other implementation.

The ISE imposes a small number of restrictions or warnings on the user. First and foremost, the entire MPI standard set of routines is not implemented. There are several of the core functions, including standard point-to-point communication, broadcasting, barrier synchronization, probing, and reduction. Any MPI function the parallel programmer uses that is not in the ISE MPI library will cause a compile-time error. Second, the programmer should place the *MPI_Init* as the very first command in the program (or as near as possible). The reason for this warning is that every process in the ISE on the local machine must lock the execution mutex

or else it may be interfering with another MPI process that is trying to take a code block timing. The *MPI_Init* is the first place in which the ISE routines can enforce this rule on the process. All computations the MPI process includes before the *MPI_Init* are computations that could possibly get included in the code block timings of another process. A similar situation is true for *MPI_Finalize*; it should be found at the very end of the user program. These function-placement limitations will be solved in the next version of the ISE.

Before the MPI processes are spawned, the *bmpi* program must spawn the local relay. First, *bmpi* creates the bridge file, local share file, and local BOnES file and initializes the semaphores, mutexes, and pseudo-semaphores in them. Next, it sets the port error flag in the share file to false and spawns the local relay. The local relay is given three command-line arguments to use for configuration: number of nodes, iteration number for which it is relaying, and socket port number to try. The *bmpi* program then unlocks the execution mutex and waits for the spawn-finish semaphore in the share file.

Once the local relay is started, it opens the bridge file and the local share file for the iteration number specified to it. Once the share file is open, the local relay has access to the execution mutex, which it locks. It then opens the BOnES file and begins to set up the socket. First, the local relay calls the *socket* function from UNIX to get a socket descriptor. Then, it attempts to bind that descriptor to the communications port number as specified to it on the command line. If the port is in use, the local relay sets the port-error flag in the share file, unlocks the execution mutex, posts the spawn-finish semaphore, and exits without fanfare. If the binding is successful, the local relay posts the spawn-finish semaphore, uses the *socket listen* command to specify the connection buffering for possible incoming connections, and unlocks the execution mutex. The local relay then waits on an infinite poll for an incoming socket connection, after which it will accept the connection.

At this point, the *bmpi* program can wait on the spawn-finish semaphore and relock the execution mutex. If the port-error flag is set, *bmpi* increments the port number it gave to the local relay by one and re-spawns the local relay in the same manner as just discussed. On the other hand, if there is no port error, *bmpi* may begin to spawn the MPI processes.

The MPI processes are spawned one-by-one in the following manner. First, the *bmpi* program places the time factor, the jump-start flag, the profiling flag, the number of nodes, and the rank number zero into the share file. It also places the name of the process machine (the machine on which it is running and on which the MPI processes will run) in the bridge file for its correct iteration. It then spawns one instance of the MPI program. These parameters cannot be passed into the MPI process as command-line arguments because the user effectively has the property rights to the command-line arguments for the parallel program. Modifying the number of command-line arguments the program can take or mixing ISE command-line arguments with user command-line arguments may very well destroy the functionality of the user's code. Therefore, no command-line arguments are added to the parallel program by the ISE and all parameters are passed in during the *MPI_Init* call. As in the case of spawning the local relay, *bmpi* then unlocks the execution mutex and waits for the spawn-finish mutex.

Once the instance of the MPI process is started, execution within the program proceeds normally until the *MPI_Init* function is reached. The MPI process then reads the bridge file with the lowest iteration number (iteration numbers appear as part of the file name of the bridge files). The process compares the name of the process machine found in this file to the name of the machine on which it is running. If the names are the same, then the MPI process has discovered to which iteration it belongs. If the names are not the same, the process reads the bridge file with the next highest iteration number. This procedure repeats until a match is found and the process knows its iteration number. This is the only method available to give the iteration number to the

MPI process without command-line arguments because the process must know its iteration number before it can open the appropriate share file to get its parameters.

Once the iteration number is known, the process can read the share file for its iteration and obtain its node number from that file. It also obtains the profiling flag value, the jump-start flag value, and access to the execution mutex, which it immediately locks. Once the process has its rank and parameter values, the *bmpi* program is allowed to overwrite them in the share file and spawn another process. Therefore, the MPI process posts the spawn-finish semaphore. However, before the process relinquishes the execution mutex, it finishes initialization by clearing the debugging time file and defining the profiling functions. The process records the time of the *MPI_Init* (which is always zero seconds) and proceeds to time its first code block. When it gets to its first MPI call, it will unlock the execution mutex, make the request to the local relay by posting the signal-to-BONeS pseudo-semaphore to the local BONeS file, and wait for a response in the form of a successful wait on the signal-to-process pseudo-semaphore. Because the local relay has not completed its initialization, the MPI process will effectively be blocked until such time as all processes have been spawned.

With the spawn-finish semaphore posted and the execution mutex relinquished, the *bmpi* program can then proceed to lock the execution mutex, spawn the MPI process with rank 1, unlock the execution mutex, and wait. The procedure repeats until all the MPI processes have spawned and come to the deadlock at their first MPI calls. The *bmpi* program then informs the user to start the BONeS simulation.

Once the BONeS simulation is started, the connection for which the local relay is waiting will eventually be made (see the below section on remote relay spawning). The local relay will then lock the execution mutex and read the machine name and process ID of the remote relay from the socket. It places this information in the local share file. If the user kills the simulation, the *bmpi* program will read this information to kill the remote relay. The user must kill the actual BONeS simulation separately using the BONeS GUI. The local relay then begins to look for communication requests from the various columns in the local share file (which certainly exist because every MPI process is deadlocked after having made such a request) and from the socket.

A.1.3. BONeS Interface Structure and Spawning

The description of the BONeS/MPI Portal itself is described in a later section of this document. This section describes the top level of the portal and how it interfaces with the outside world. The interface is completely contained within C functions that are called from BONeS via BONeS's primitive block ability. When BONeS starts a simulation, multiple-iteration or not, it numbers the iterations beginning from zero. BONeS provides a runtime block that outputs the iteration number. It is this block which is used to feed the primitive functions the iteration number so that the correct files can be accessed. There is no such BONeS block to automatically provide the node rank number to the primitive functions in the individual nodes. Instead, the node rank must be designed into the user's network model and passed to the portal as an argument. The portal then passes the rank number to the primitive functions.

When the BONeS simulation is started, for each iteration, a BONeS "Init" in the system view of the iteration is activated. At this initialization, the BONeS simulation reads the bridge file for its iteration, creates the share file on the remote (BONeS) machine, creates the remote BONeS file, and spawns the remote relay. The command-line arguments to the remote relay include the name of the machine to which it should connect (obtained from the bridge file for the iteration), the port number to which it should connect (also from the bridge file), the number of nodes, and the iteration number. The BONeS simulation for that iteration then waits on the spawn-finish semaphore in the remote share file. The processes on the remote machine do not need to lock the

execution mutex in the remote share file because there is no code-block timing happening on the remote machine.

When the remote relay begins, it reads the bridge file, the share file, and the BONEs file for its iteration. It then uses the UNIX *socket* command to get a socket descriptor and the *connect* command (with the agreed-upon port number) to connect to the local relay. Once the connection is established, the remote relay informs the local relay of the name of the remote machine and the process ID of the remote relay. This information will be used by *bmpi* to kill the remote relay if there is an abnormal simulation termination on the user's side. The remote relay then posts the spawn-finish semaphore in the remote share file for its iteration and begins to look for communication requests. Requests from the MPI processes to BONEs will arrive on the socket and communication requests from BONEs to the processes will appear in the various columns of the remote BONEs file. The first communications will always come from the processes because BONEs is waiting for them to make a request.

Once the "Init" block in the BONEs system view for a particular iteration has run the primitive to complete the above operations, the portal at each node is then initialized. This initialization includes reading the bridge file, remote share file, and remote BONEs file. Each node then proceeds immediately to retrieving the first request from its counterpart node on the local machine.

A.1.4. Communication and Relays

This section presents the procedures used by the elements of the ISE to complete the various MPI calls. Included are detailed descriptions of how the MPI processes, relay programs, and BONEs simulation interact to complete the *MPI_Send* and *MPI_Recv* function calls. Due to the fact that the methods used to implement the remaining MPI functions are variations of the methods for the send and receive, the descriptions of the remaining functions are shorter. To help in the understanding of the detailed operations, example situations are used throughout the text.

Each MPI process keeps a private copy of the current time of the simulation in its memory. At initialization, each process sets this current time to zero. Assume the first MPI call for process 3 is an *MPI_Send*. After initialization, process 3 would lock the execution mutex and time the code block that precedes the *MPI_Send* call. The time for the code block, assumed to be 2 seconds, is then added to the private copy of the current simulation time, in this case yielding a new simulation time of 2 seconds. Then, a "send overhead" is added to the simulation time, say 0.1 seconds. This "send overhead" is obtained on the fly by timing some code that functionally does nothing but computationally would do the same things required for an MPI implementation to issue an *MPI_Send*. That is, the added overhead simulates the time the MPI implementation, not the user program, spends in setting up a send call.

Process 3 then unlocks the execution mutex and makes the request to send. To issue the request, the process places the time of the call (2.1 seconds) into the request structure field in the local BONEs file under column 3 (for process 3). It also places an integer code for the type of call (an *MPI_Send*) into the request structure. The process then locks the signal-to-BONEs mutex in the share file and increments the signal-to-BONEs integer in column 3 of the local BONEs file to 1. Finally, the process unlocks the signal-to-BONEs mutex. The entire procedure of locking the signal-to-BONEs mutex, incrementing the integer, and unlocking the same mutex describes the entire process of posting the pseudo-semaphore for the column-3 signal to BONEs. It is important to note that the MPI process must unlock the execution mutex before posting the pseudo-semaphore because the locking sub-operation at the beginning of the procedure may block the process. Blocking the process that holds the execution mutex may cause deadlock

when no other process can execute. After the post, process 3 waits on the signal-to-process pseudo-semaphore.

The local relay program is continually looking through the local BONEs file to see if the signal-to-BONEs pseudo-semaphore has been posted in any of the columns. In order to do this polling, the relay stays in a loop in which it continually issues *try-waits* on the pseudo-semaphores. The entire try-wait procedure is composed of the following sub-operations. First, the relay tries to lock the signal-to-BONEs mutex in the share file. If it is not successful immediately, the mutex try-lock returns, as does the pseudo-semaphore try-wait; the result is simply that the pseudo-semaphore is not yet posted and it will try again later. If the mutex try-lock is successful, then the relay program has the lock on the signal-to-BONEs semaphore. It checks the value of the signal-to-BONEs integer in the column of the process it is polling. If this value is 1, the relay sets the integer to 0 indicating the wait, unlocks the signal-to-BONEs mutex, and returns a true flag from the pseudo-semaphore try-wait call. The relay program then knows that the process for that column is trying to communicate. If the integer is still 0, then the relay unlocks the signal-to-BONEs mutex and returns false from the call. The relay program would then try either the next column in the same manner or the socket.

Returning to the example, the local relay soon finds that the try-wait on column 3 is successful. It reads the call type and the time at which the call is to take place. The relay repackages this information in a fixed-size relay packet. Also included in this packet is the signal type set to "send no data" and the process rank. The local relay sends this packet into the socket and returns to polling the signal-to-BONEs pseudo-semaphores and the incoming socket.

The remote relay is also in a loop polling the socket and the remote BONEs file for signal-to-process pseudo-semaphores. The remote relay eventually finds something on the socket and reads from the socket for the known length of the relay packet. Once read, the signal type in the packet ("send no data") informs the remote relay that there is nothing more to read from the socket for this particular transaction. The remote relay reads the process rank (3) and places the call time and call type in the request field in column 3 of the remote BONEs file. It then completes a post to the signal-to-BONEs pseudo-semaphore in the remote BONEs file.

At this point, the BONEs simulation is waiting for the posted signal from the remote relay. It is important to note that the BONEs simulation does not proceed past the time of the last completed MPI call until it knows the next request time for every process. That is, no simulation time can proceed until BONEs knows at what time to stop for the next MPI call. Had the ISE not been implemented in this manner, it would have been possible for simulation time to pass in the simulation and for a request to arrive after it should have been serviced.

Node 3 of the BONEs simulation waits on the signal-to-BONEs pseudo-semaphore. It then reads the call type and call time for the requested MPI operation. These values are stored in a local BONEs variable in memory. BONEs posts the signal-to-process semaphore in column 3 of the remote BONEs file to acknowledge receipt of the request. The BONEs node then sets a timer to trigger the portal block at the correct request time.

The post that the BONEs node made is propagated through the remote relay, socket, local relay, and finally the local BONEs file. The MPI process waits on this semaphore. Once the process knows BONEs has received its request, it writes the *MPI_Send* information into the local BONEs file. This information includes the amount of data to send, the destination of the data and the data tag. The signal type field is set to "send data." Process 3 also places the data it wants to send into the file and posts the signal-to-BONEs semaphore. If the data is too large to fit in the BONEs file, the piecewise transfer previously discussed is used.

The local relay sees this post and reads the information. It places the destination, tag, size of data, and signal type information in a fixed-size relay packet, which is sent through the socket. The local relay also reads the data into its memory space. The address where it stores this data is then also sent through the socket. The actual data vector itself does not leave the local relay's data space. At a later point in time, when a receive is completed, this address will be consulted to retrieve the data. This procedure is used to optimize the operation of the BONEs simulation because BONEs would have needed to manage a large data structure through its many complicated blocks if the actual data vector was sent. Instead, BONEs must only propagate the 4-byte address of the data.

When the remote relay sees activity on the socket, it reads a fixed-size relay packet from the socket and discovers that there is additional information to be read (via the "send data" signal type). It reads the local relay's address of the data, places all the information in the remote BONEs file, and posts the signal-to-BONEs semaphore in column 3. At this point, BONEs is still simulating the time up to the request time. When BONEs reaches the request time, it reads the information, creates a packet out of the destination, data size, tag, and local relay address, outputs the packet on its simulated network, and posts an acknowledging signal-to-process semaphore.

This acknowledgement will proceed through the message path just as described before. When the MPI process waits on the semaphore, it locks the execution mutex and proceeds to time its next code block to repeat the entire process.

Now suppose process 2 wants to receive a message from the network. In the same manner as described for making an *MPI_Send* request, process 2 issues a request to BONEs indicating the type of call (*MPI_Recv*) and the time of the call. Suppose this time is 3 seconds. After receiving the acknowledgement, the process sends the source from which to receive (which may be *MPI_ANY_SOURCE*), the tag of the message to receive (which may be *MPI_ANY_TAG*), and the expected size of the data to BONEs. The process then waits. When the request time arrives, BONEs reads the desired message envelope and searches its buffers for a message that has already arrived that matches the envelope.

If it finds the message, which is comprised only of an envelope and the local relay address, then it sends the information through the message path to the local relay. BONEs also sends the current time, which is 3 seconds because the desired packet was already available at the time of the MPI call. The local relay receives this information and places it into the local BONEs file in column 2. The local relay also accesses the address in its memory space. It places the information at this address into the BONEs file. The local relay then signals process 2, which reads the data into its memory space. In addition, the process reads the completion time of the operation and adds a "receive overhead" (say, 0.1 seconds) to the time. This receive overhead is obtained in the same manner as the send overhead previously described. The process then overwrites its local copy of the current simulation time with this new time (3.1 seconds).

On the other hand, if the BONEs node does not find a match to the requested envelope immediately, it will not respond to the process and will continue to simulate the network. As time in the simulation passes, the desired packet will arrive and BONEs will follow the steps outlined above to inform the process of the arrival. This situation is the reason why BONEs must inform the process of the completion time of the operation, for without such a mechanism, the process would have no idea at what time the packet was received and its request times for later MPI calls would be erroneous.

There are two points that need elaboration about the operations the local relay carries out to move the data from its memory to the memory of the process. First, recall that the address received from BONEs will point to a data location. The local relay will also receive information about the length of this data. If the data is longer than there is room in the BONEs file, the

piecewise transfer used in the *MPI_Send* is used in the opposite direction. Second, once the data has been safely and completely transferred to the MPI process, the local relay deallocates the memory where it was stored. An exception to this rule is that the local relay cannot delete data that was sent via a broadcast until all the receives have been completed.

With the above operations for the *MPI_Send* and *MPI_Recv* functions in mind, the operations of the other MPI calls included in the ISE are simple in nature. For a broadcast, the root node's operations are the same as for *MPI_Send* except the destination node is set to "-1." The nodes that are not the root issue *MPI_Recv* operations from the root. Again, the relay must be careful not to delete the data before all receptions are complete. For *MPI_Probe*, the *MPI_Recv* operations are followed except for the fact that the packet remains in the portal and only the matching envelope information is sent to the process. For the *MPI_Iprobe*, the *MPI_Probe* operations are followed except that the portal will always acknowledge immediately with a true or false instead of waiting for a matching packet to arrive. For *MPI_Barrier*, the nodes participate in an all-to-all communication using broadcasts. When a node has received a barrier packet from all other nodes, the synchronization has been completed. The *MPI_Reduce* call uses standard sends and receives to get the data vectors to the root, which then executes the desired reduction operation. The *MPI_Allreduce* function simply adds a broadcast to the end of the *MPI_Reduce* so that all nodes will receive the answer to the reduction operation.

A.1.5. Portal Internals

This section examines in detail the internal operation of the Portal block, which is attached to the top of the user's fine-grain network model. This Portal contains the primitive code which interacts with the remote relay and remote BONEs file, as described above. However, the Portal also contains several blocks that are used to handle the MPI requests once the primitive code reads the necessary information. These blocks are described in this section.

A.1.5.1. Decomposition of ISE Functions

The ISE Portal is composed of many primitives that are mostly contained inside of one block function called *BONEs_MPI Portal*. The block diagram of this function is shown below in Figure A.4. This function contains the primitive functions *BONEs_MPI Initialize*, *BONEs_MPI Get Request*, *SCALE Input Queue/ Recv Functions*, *BONEs_MPI WTime*, *BONEs_MPI Barrier_time*, *BONEs_MPI Trigger_Network*, *BONEs_MPI Send*, and *Set Termination Flag*. The remaining numerous blocks are part of the BONEs Core Pool Library and are not discussed in full detail.

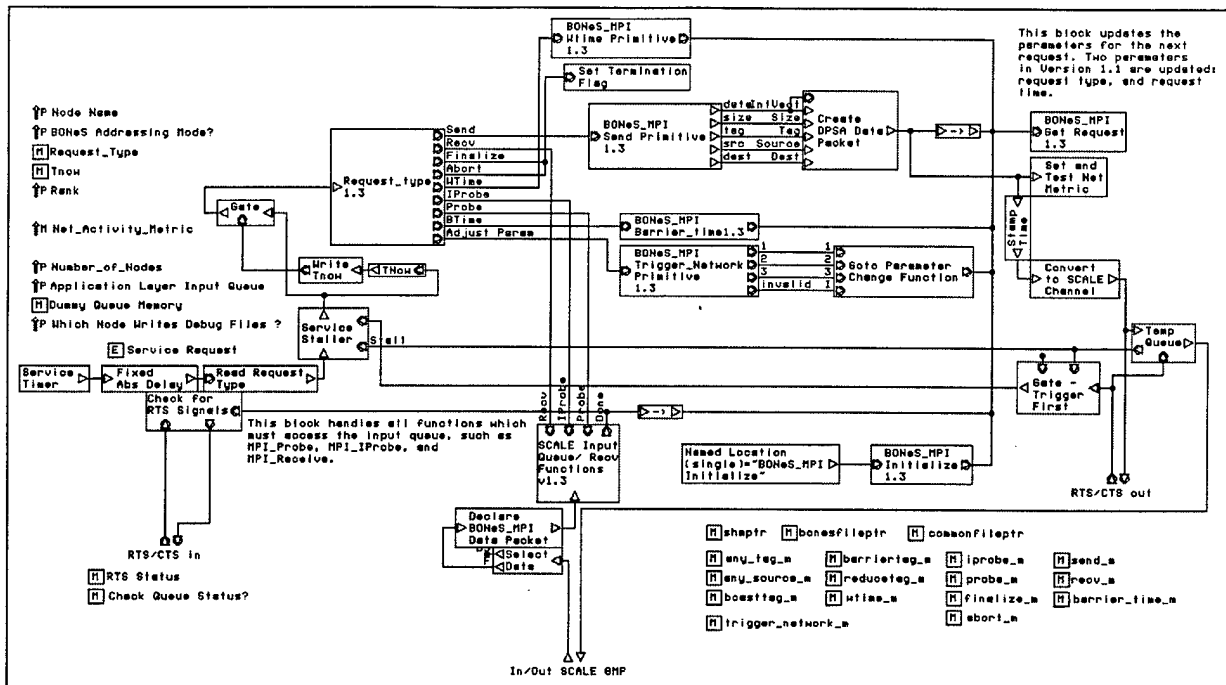


Figure A.4 - The BONEs_MPI Portal

The BONEs_MPI Portal is essentially the gate between the MPI processes and the BONEs simulation environment. It contains three ports to link into a network simulation: a bidirectional data port, and two sets of RTS/CTS (Ready to Send/Clear to Send) signaling ports.

The *BONEs_MPI Initialize* block is a function that is executed once at initialization. However, BONEs allows orders in initializing blocks, and it was particularly important to execute the *BONEs_MPI Initialize* block before any other block was allowed to function. As was discussed previously, ISE uses shared files to communicate between the MPI processes and the BONEs simulation. In order to write to these shared files, each node has to query the UNIX kernel for a file descriptor pointing to the named file.

The *BONEs_MPI Get Request*, in the upper right-hand corner of Figure A.4, queries the shared files (as described above) for its current task. BONEs will stall until a request is placed by the process, so the process must ensure that a request is available to BONEs whenever it has completed its last task. This was also discussed in further detail above. The *Get Next Request* block sends out two statistics. The first is the request type and the second is the request time. These statistics tell BONEs what it is going to do next and when it is going to do it. The Portal places an event at a future time with event timers. When this time has expired, a trigger is routed via the request type number to the requested block function, which may be one of nine requests: *MPI_Wtime*, *MPI_Send*, *MPI_Receive*, *MPI_Probe*, *MPI_Iprobe*, *MPI_Finalize*, *MPI_Abort*, *Barrier Time*, or *Adjust Parameter*.

The *MPI_Wtime* is used to inform the MPI process of the current simulation time. Note that when this request is serviced (as is done for any other type of request), that a trigger is sent back to the *Get Next Request* function, except in the case of the *Set Termination Flag*.

The *Set Termination Flag* block handles the event *MPI_Finalize*. Its primary function is to shutdown the simulation. Because of this, it does not require a trigger to the *Get Request* block. Essentially, all of the requests have been consumed and that node is ready to shutdown. This function, unlike most of the ISE functions, is not a primitive, but is made up of modules that

BONeS provides. The *Set Termination Flag* works with the *Test Simulation End* primitive, which will terminate the simulation when each of the nodes has set this flag. The *Test Simulation End* is not part of the ISE Portal function because it must be placed at the system level.

The *MPI_Send* is one of two essential operations (send and receive) for a message-passing paradigm. The send function executes the *Create DPSA Data Packet* block using the envelope information it has retrieved from the BONeS file. Notice that the structure is composed of five main components: an address specifying the data location in the local relay, the data size field (in bytes), a tag field, the destination, and the source. The tag and the source are the basic discriminates in receiving a packet. The destination field is used primarily to route the data packet through the network. The data may be of any arbitrary size. Any data types may be sent through the network (i.e. integers, floats, characters, etc.) because only the size is important (the local relay has already dealt with storing the data in memory space).

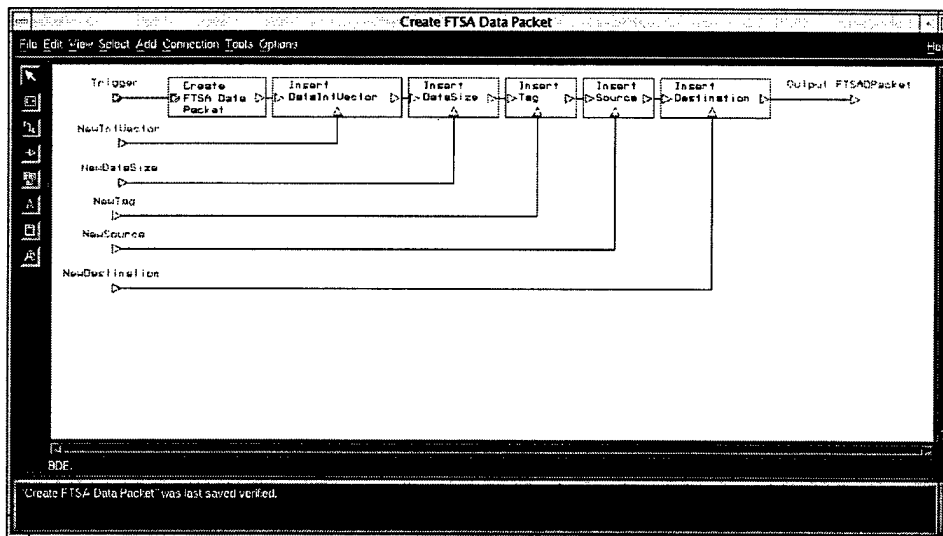


Figure A.5 - Create DPSA Data Packet Function. This function creates the data structure for the BONeS_MPI Data Packet.

The request type *MPI_Recv* is another essential function for a message-passing interface. The *MPI_Recv* block and the probe blocks (for *MPI_Iprobe* and *MPI_Probe*) are the only BONeS/MPI primitives with two services. Note that there are two inputs into the device. The trigger input is serviced when the request time expires. It reads the message envelope information from the shared file and sends the tag and source fields to the two outgoing ports. The majority of the blocks in the *SCALE Input Queue/ Recv Functions* are used in correctly receiving MPI data packets. Each of the incoming data packets are placed into a buffer called *FIFO w/Reset and Length*. This FIFO temporarily holds onto incoming packets until they are matched with the same source and tag fields output from the two ports of the *MPI_Recv* block. When a match is made, the data packet flows through a switch until it is sent into the input data port of the *MPI_Recv* module. The *SCALE Input Queue/ Recv Functions* block (shown below as Figure A.6) includes the *MPI_Recv*, *MPI_Iprobe* and *MPI_Probe* functions and the *SCALE Test Incoming Packets* block, which provides the tag and source-matching functions.

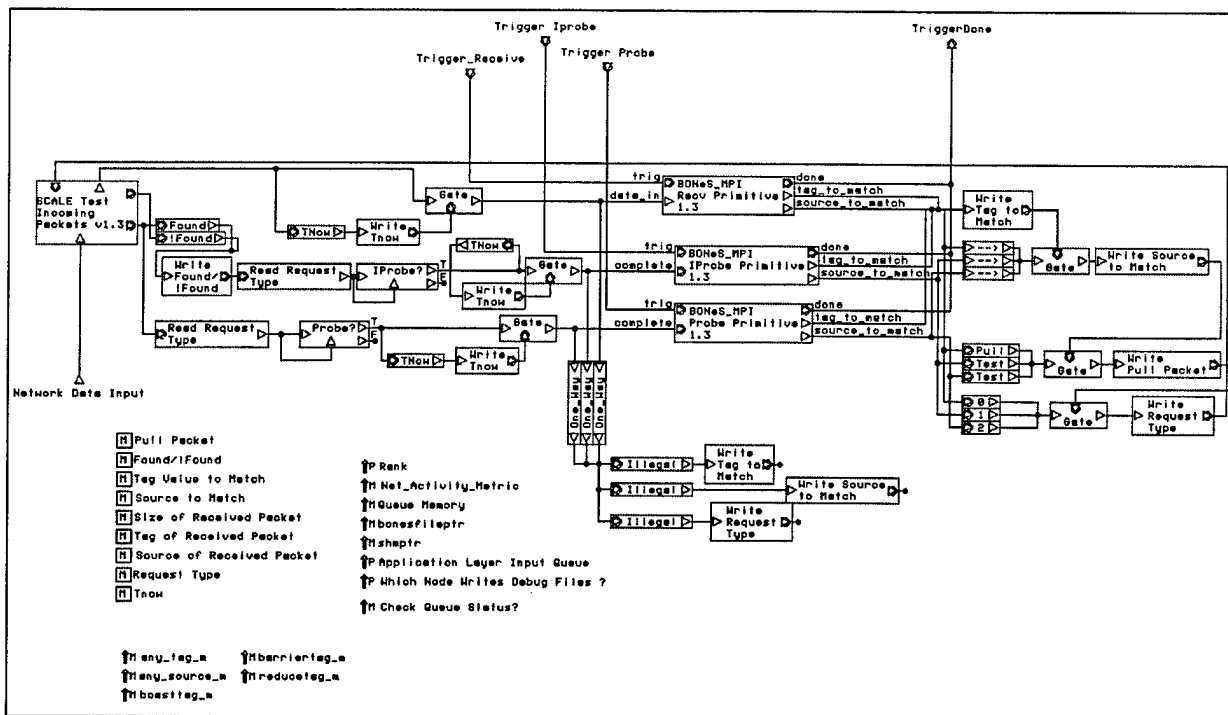


Figure A.6 - Close-up of Receiver Matching Logic.

The ISE must buffer incoming data packets until they are matched with a corresponding receive request. The circuit above shows the logic for matching two parameters: *tag* and *source*. This module requires multiple arguments that are stripped off of the packet along the way. These arguments include the source, the data size, and the tag. The current model also supports *MPI_ANY_SOURCE* and *MPI_ANY_TAG* fields. The second service of *Receive Request* is to write the matched message envelope and the local relay address to the BONEs file and trigger the *Get Request* module. The other functions of the ISE Portal are described in further detail in the Implementation Guide section below.

A.1.5.2. Enhancing ISE Performance

One of BONEs' greatest limitations is its computational requirements, especially when simulating a high-fidelity model with many nodes. Simulating one second could take thousands of hours. Since this is impractical, a few modeling tricks were employed to speed up the simulation time. Many methods were proposed to combat this problem including reducing the fidelity of the models. However, since this is, in general, a poor solution to the problem, a more robust alternative was necessary. The technique employed was to disable parts of the model, in effect sending them into a waiting state. The philosophy is this: if the network is idling, lower the fidelity of the model temporarily. The particular models that were tested for this were driven by a master clock. Every aspect of each model was controlled by this clock device, much like a real electronic device. This provided the perfect point to place a shutdown gate. Thus, the model operates in two modes, regular mode and standby. Unfortunately, in some programs, situations in which the standby mode can be incorporated may be quite rare, so speedup of simulation time may not be gained. However, in the procedures tested, there was ample opportunity for speedup in standby mode. Standby mode occurs only if the entire network is idling. If any single node has any piece of data on the network, the model has to operate in regular mode. To account for traffic on the network, a single global variable was created that could only be incremented and

decremented. When a piece of data enters the network from the Portal, this value is incremented; when it leaves, the variable is decremented. In this manner, an exact account of how much traffic is currently in the network is known. In addition, each node must take into account the next time it needs to service a request. In the case in which there is no traffic on the network and every node will service a request in the future, the simulation can basically fast forward to the earliest time for the next request without hurting the fidelity of the model. A pair of functions is used to control this functionality. Figure A.7 below shows the *Global Clock Gate* function which sets the *Global Next Event Time* when the simulation can fast forward. This variable is set to the nearest service time. The complementary function *ClockRouter* is used to route the clock signal when *TNow* is greater than the *Global Next Event Time*, otherwise the clock signal is sunk. Figure A.8 shows the master clock with the *ClockRouter* gate on its exit path. In fast-forwarding mode, only the clocks and fast-forward mechanisms are simulated, which greatly affects the computational time in simulations with lightly loaded networks.

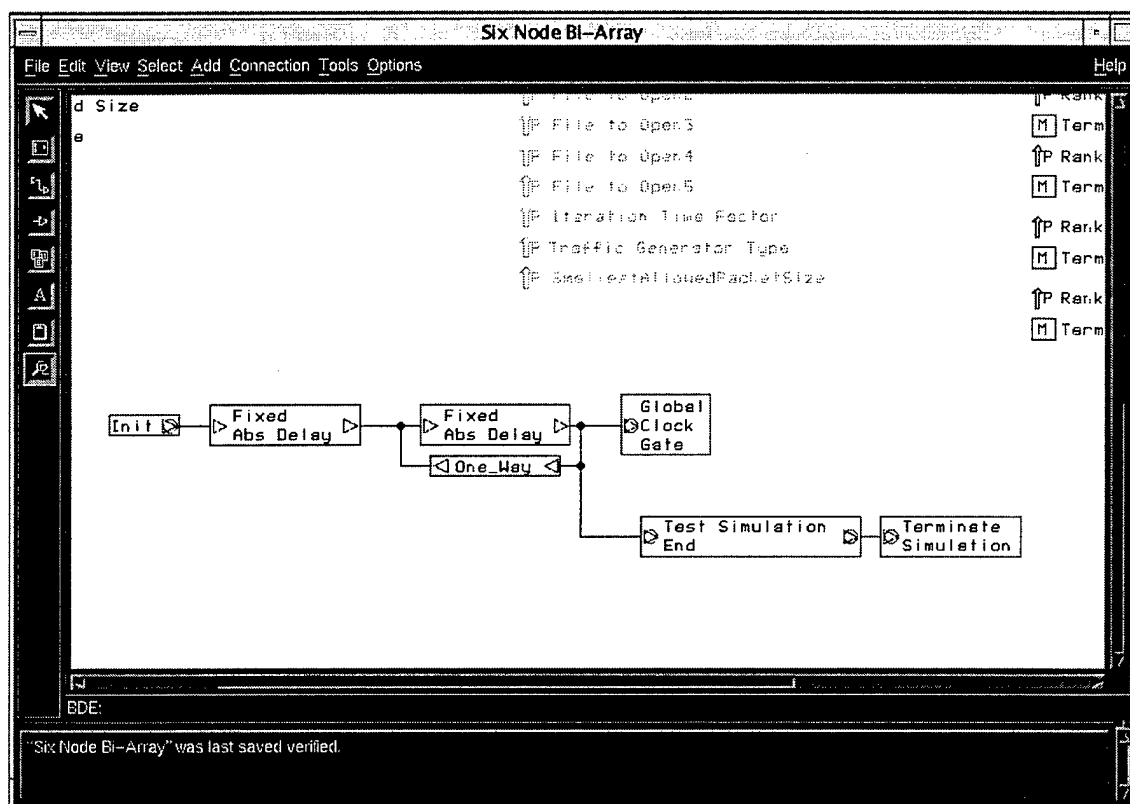


Figure A.7 - Close-up of the Six-Node Bidirectional Array's Global Clock Gate. The Global Clock Gate tests whether or not the simulation can fast forward to a later time. The system may lower undesirable computational time by reducing the fidelity of the model during idle periods.

2. Make sure that the ISE package directory is included in the path variable located in `.cshrc`.
 3. Run `ise` to begin the program.
 4. Choose the directory that contains the parallel program.
 5. Specify the number of iterations that will be executed. Multiple iterations occur when multiple values are assigned to a single parameter in BONEs (also known as range variables). If unsure, specify one iteration.
 6. Specify the number of nodes on which to simulate processes. That is, if the BONEs network consists of 5 network nodes, then 5 nodes should be specified here.
- Steps 7 – 10 should be repeated for each iteration specified above.
7. Select the type of machine that will simulate the processes. These categories come from a specially formatted `.rhosts` file located in the user's home directory. An example format can be found in the appendix. If this format is not used, only one category will appear at this step.
 8. Select a machine of the given type that will handle all of the parallel processes for this iteration. This machine must be different from the one used to run the ISE and BONEs, hence the need for two machines at a minimum is illuminated. After selection, a list of the most taxing processes on that machine will be displayed. The machine should have a relatively light load to produce accurate results. If the machine seems too heavily loaded, the selection may be rejected and a new machine may be chosen for this iteration. Each new iteration will require a new machine and the ISE will prevent selection of the same machine more than once.

NOTICE: Failure to properly select machines to simulate the parallel processes may provide inaccurate results or prevent the simulation from starting.

9. Enter the command line of the parallel program followed by any command line arguments used by that program. Notice that unlike other MPI implementations, no MPI-specific command line arguments need to be included.
10. Enter a time factor that scales the relative performance of the processor. For example, if the processor actually simulating the processes is twice as fast as the processor being simulated, enter a time factor of 0.5.
11. Activate profiling if parallel analysis after simulation execution is desired.
12. To decrease simulation time, jumpstart may be activated. This feature makes the BONEs simulator "jump" to the first MPI call. This feature may be deactivated at this time to prevent simulation crashes.
13. The output produced by the MPI program itself may be redirected to the file entered at this prompt. This feature may also be deactivated at this time.

The following prompts appear after the above steps have been followed for every iteration.

14. All of the settings specified in this session can now be saved. The user then has the option of leaving the program. Continuing the program will start the parallel processes and spawn a new terminal window for each iteration.
15. The ISE is now ready for the BONEs simulation to begin. The terminal windows that were automatically spawned should list the size of the model that the processes are expecting.

A.2.4. Starting the Network Simulation

1. Open the model library containing the highest-level model of the system. The network structures and BONEs/MPI components should have already been connected and constructed to some arbitrary size (i.e. number of nodes).
2. Open the "BONEs_MPI Pool 1.3" library. This contains the necessary primitives for the BONEs/MPI components to communicate properly with the ISE.
3. Go to the BONEs Simulation Manager and select or create a simulation based on the correct number of nodes. This number should be equal to the node number specified in the ISE.
4. The number of iterations must also match that specified in the ISE. However, this number is not set explicitly in BONEs. Rather it is specified through the use of range variables in one of the run-time parameters. For example, if two different packet sizes are specified in the packet size parameter window, then two iterations will be simulated. When using a simulation that was already created, it is a good idea to check all the parameters to make sure there are no range variables. If more iterations are started on BONEs than by the ISE, the simulation manager will likely crash the simulation.

5. After all of the parameters have been set correctly, open the Run dialog box.
6. Select the machine(s) that will be chosen randomly by the simulation manager to run the iterations. For the same reasons as given previously, none of these machines should be the same as the machines chosen from within the ISE to simulate the processes. If unsure, a safe, albeit slow, choice would be to select the same machine that is currently running BOnES and ISE to also run the simulation.
7. Ensure that the pool selected is the BONES_MPI Pool 1.3.
8. Set TSTOP to 1,000,000. This will prevent the Simulation Manager from stopping the simulation before the MPI program has finished executing. If this value is not high enough, increase it. Note that doing this will render the progress indicator during simulation useless.
9. Once all options in the Run dialog box have been set correctly, start the simulation. If there are questions about various options, refer to the BOnES documentation.

A.2.5. After the Simulation Has Started

1. As the simulation runs, printf statements from the parallel program will display in the respective iteration windows. The total execution time is best reported by using timing functions from within the parallel program as if being run on the actual hardware. This method will also make the code more useful and portable.
2. If profiling was selected earlier in the ISE setup, a freeware program called Upshot will now be spawned. The computation and communication patterns recorded in a log file during the simulation are displayed.

A.2.6. Tips for Advanced Users

- The ISE and BOnES may be run on separate network machines by activating xhost on the local machine and setting the DISPLAY environment variable correctly.
- If too few iterations were started by the ISE and BOnES hasn't started yet, additional process iterations may be started by hand. The bmpi program was actually called repeatedly by the ISE but can be executed by the user. Type bmpi at the prompt for a list of run-time options.

A.2.7. Requirements of the MPI program

- The program should not perform any operations that need to be timed before the *MPI_Init* or after the *MPI_Finalize* functions have been called.
- Not all of the functions included in the MPI specification are supported. A list of supported MPI functions can be found in the appendix.

A.3. Implementation Guide

This section introduces the information required for a user to integrate a network model into the ISE. It also serves as an introduction to the BOnES/MPI environment and what tools it offers.

A.3.1. Portal

The BOnES/MPI Portal (so named because it is the doorway in BOnES to the MPI software previously described) may be thought of as the application layer of an OSI network protocol stack. It essentially offers services to MPI processes. Since MPI offers error-free communication, it is the job of the modeler to provide a fully functional protocol stack including segmentation and reassembly (if needed), error handling, and flow control. Without these components MPI will certainly fail. This is especially true when interfacing the ISE to a lossy

network such as ATM or when modeling networks with fault-injection. Figure A.9 below shows a protocol stack for a bidirectional array and next to it a protocol stack of the SCI network.

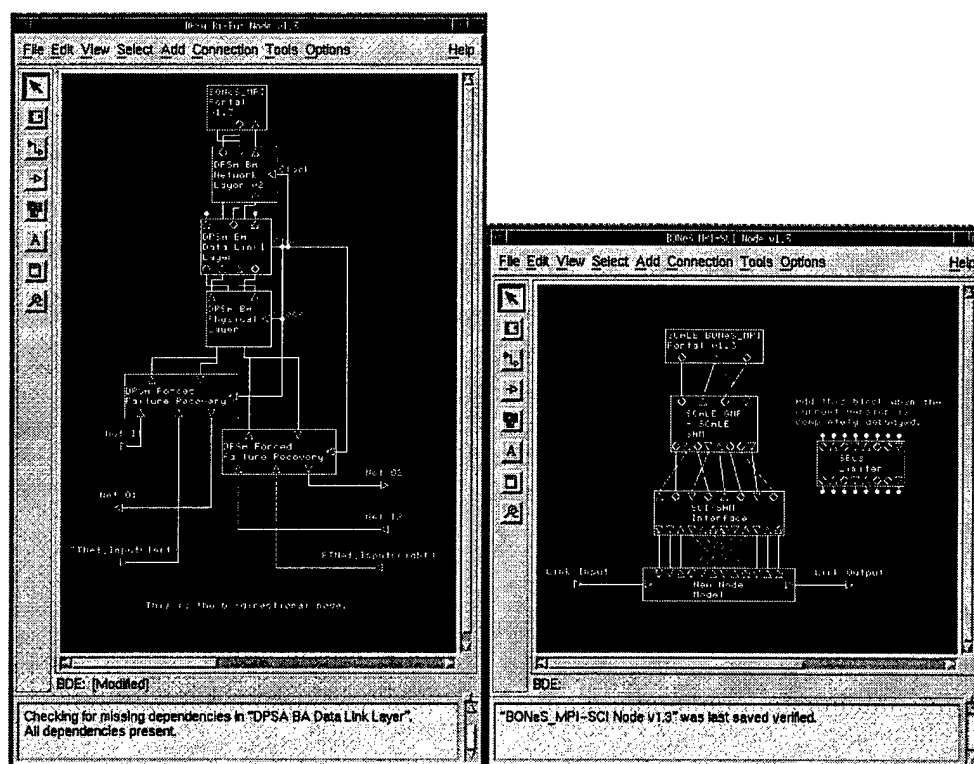


Figure A.9 – BONE_S/MPI Portal over Two Network Protocol Stacks

There are essentially three functions of the BONE_S/MPI Portal: scheduling events, performing actions, and queuing and matching incoming messages to process requests. Each time an MPI process requests a transaction with the BONE_S/MPI Portal, it sends both the transaction type and when to schedule the transaction. BONE_S then schedules the action to occur with a timer. The portal currently can schedule nine types of events: sends, receives, finalizes (shutdown network), aborts (abort connection and close network), request of network clock time, probe (check queue and wait), iprobe (check queue and return), barrier time (useful to setup up real-time systems), and trigger network parameter. This does not include other services offered for MPI processes, which are macros of these network services (including some important calls such as *MPI_Barrier*).

The actions correspond to each of the scheduled events. The two main self-explanatory services to the network are sending and receiving. A few of the actions are not supported by the MPI standard and thus require some explanation. The first of these is the barrier time mechanism. As mentioned previously, this helps schedule events at certain event times for real-time systems. For instance, the event may be the generation of a new data stream from a data acquisition board. If these occur at regular intervals the time may be clocked by the network interface. The other BONE_S/MPI-unique function is *Trigger Network Parameter*. This function may be useful in setting up multiple pseudo-iterations within the same program and within the same BONE_S iteration with different network parameters (provided that the BONE_S parameters are actually memory arguments). For instance, the user may want to run the different pseudo-iterations at different network bit rates. The trigger network block actually sends a signal to the

system layer where the “BMPI Functions” block is located, so the parameter change blocks must reside in the system view. These functions are discussed in further detail below.

The three actions receive, probe and iprobe each access the portal’s input queue. This set of blocks matches incoming messages to those that MPI is trying to receive. It also supports wildcards such as *MPI_ANY_TAG* and *MPI_ANY_SOURCE*. Figure A.10 below shows the innards of the block that supports these services.

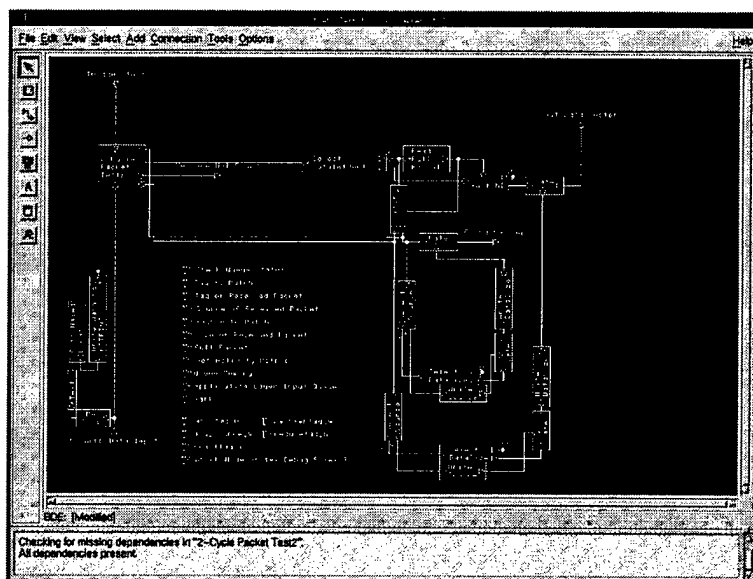


Figure A.10 – Portal Input Queue

The system block (shown in Figure A.11 below) includes the mandatory system-wide block that contains four functions: initializing of the communication link (which communicates to the processes), receiving the signals for the trigger parameter change (a total of three), shutting down the BONEs simulation (via *MPI_Finalize* or *MPI_Abort* from the MPI processes), and lastly speeding up simulation time by skipping over periods with no network traffic. This block is required in the system diagram. Without it, the communication link to the processes will not be made. The other three functions are convenience features. The block that initializes the communication link spawns off the remote relay program as discussed above. Recall that the trigger network parameter mechanism may be used to change an argument or the state of a BONEs simulation. When an MPI process calls *MPI_Finalize*, it sends a signal to the system functions block. Once each node has sent this signal, the simulation will be terminated. The *MPI_Abort* call will also be supported in the future to shut down the ISE environment. The Fast Forwarding mechanism is the most difficult to integrate into a network model but is optional. The user may integrate a fast forwarding device into the model to speed up simulation. The network accounts for all of the MPI messages in the network by counting all of the messages leaving and all of the messages that have returned. Broadcast messages are also accounted for by incrementing the counter by the number of nodes (less one because the sender does not receive the message). To integrate this capability into a model, the user may use a linked piece of memory called “Global_Next_Event_Time”. When there is nothing to send on the network, this variable is set to a real time of infinity (actually 10000.0 which is much longer than any simulation is likely to be). When a message is on the network this value is set to 0.0. The value of this variable may be used as a gate from a timer. If the user requires a signal to be sent from this block after the simulation has been disabled for a period of time, the conclusion should be drawn that a “Named Goto” block (one of BONEs core primitives) with a string argument of

“Start Clock”. When the BONEs_MPI portal sends a new message onto the network, a signal will be generated to these named Goto blocks.

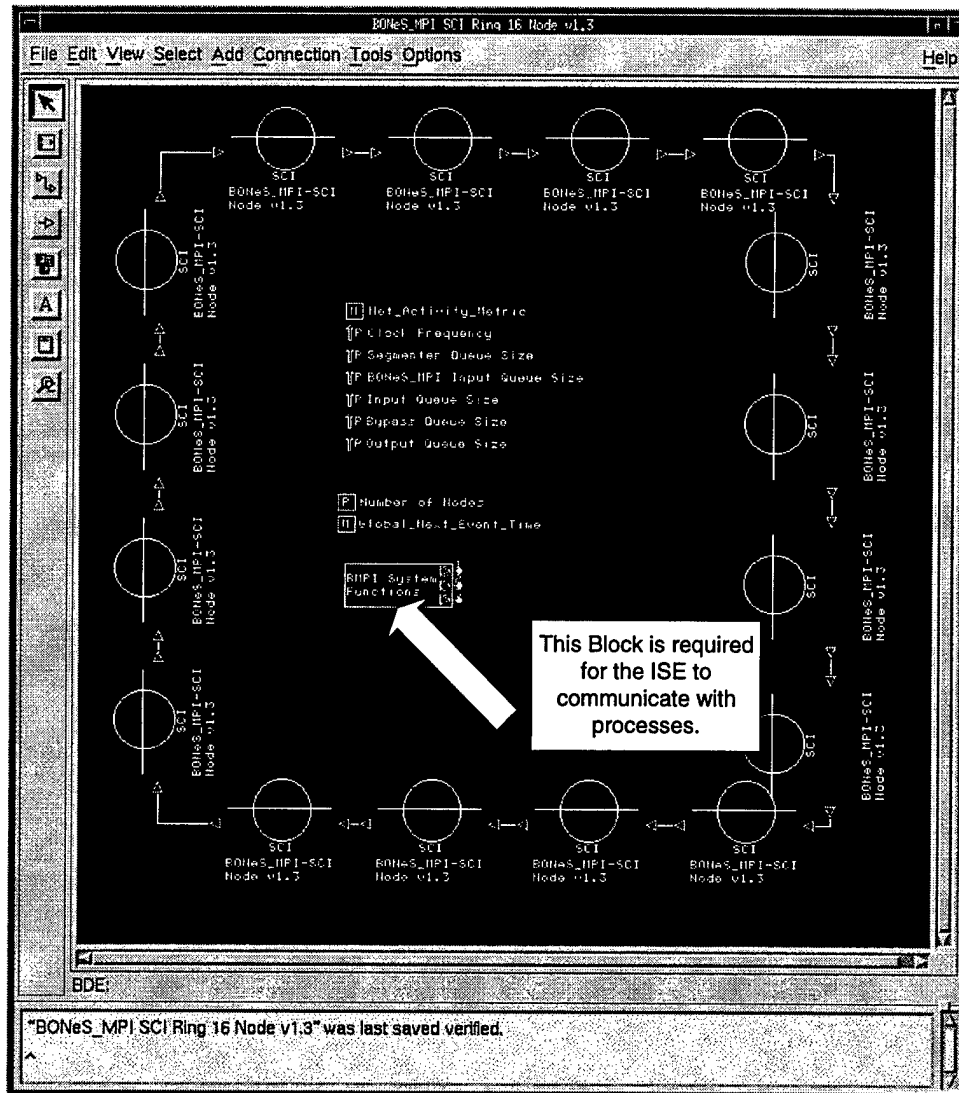


Figure A.11 – System View for an ISE-Capable Network

A.3.2. Hooking up the Portal

Currently there are two versions of the BONEs_MPI Portal function and both are identical except for the connection interface. The reason for two portal versions arose because there was want of a BONEs_MPI Portal that operated in an HCS standardized model called SCALE GMP (Scalable Coherent Architecture Latency-hiding Environment, Generic Message Passing). The interface in the SCALE GMP version is based on signaling principles likely to be implemented in real hardware. Figure A.12 below shows both the SCALE GMP version “SCALE BONEs_MPI Portal v1.3” and the older version “BONEs_MPI Portal v1.3”. It is recommended that the modeler use the SCALE GMP portal since it is more powerful and generic. However, both are discussed for completeness.

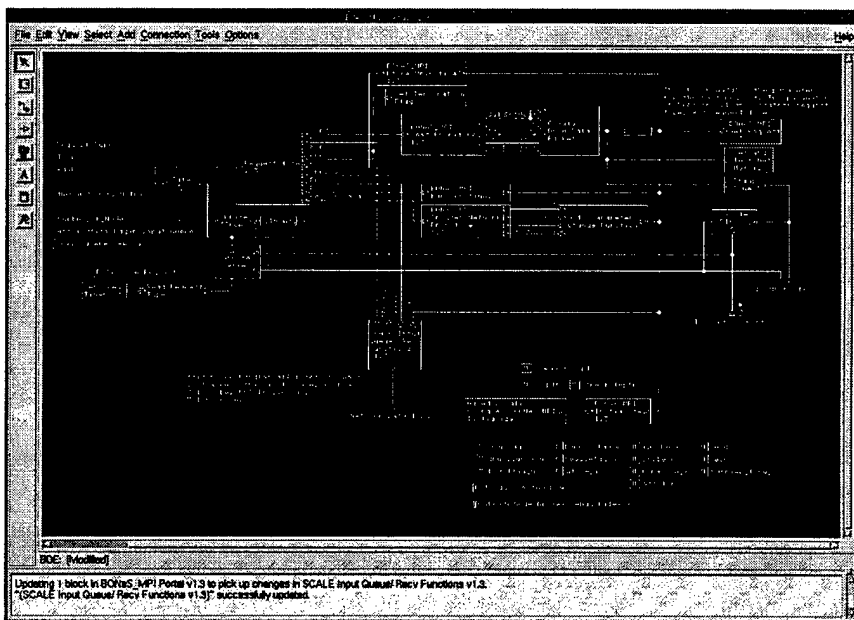
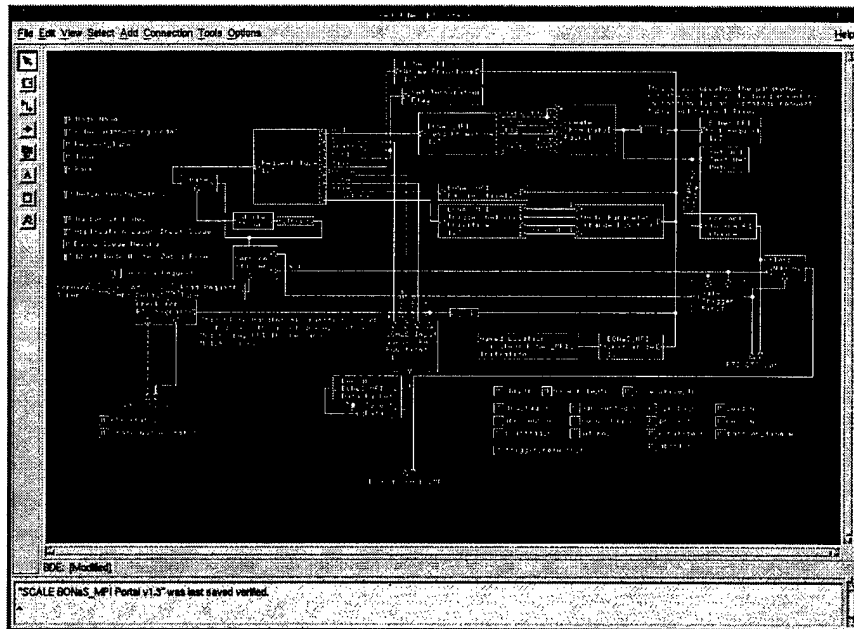


Figure A.12 – Internals of the Two Versions of the Portal

The original, BONEs_MPI Portal v1.3, has only four ports: a send port, a receive port, a network-reject port, and a network-ready trigger. This protocol does not allow for rejections from the portal layer. If it is unable to receive it will simply drop the incoming data. Unfortunately, this situation causes a lockup. However, most models will not be able to handle a reject anyway since reverse flow control would be required. The send port works in conjunction with the remaining two ports: network-reject and network-ready. The network can reject incoming messages and send them back into the portal. The BONEs_MPI portal layer is then stalled until a network-ready trigger is received from the lower layer. This implies that the network model must ensure that a ready trigger is sent out if a rejection occurs. The data ports (all but the network-ready trigger port) use the structure “BONEs_MPI Data Packet”. The fields of this data structure are shown below in Figure A.13. The user may parse the five fields to pull off any information

that may be needed by the network model such as destination and data size. However, the entire data structure must be encapsulated into a “root-object” data field before moving it through the simulated network. This is because the receiving node must move the information back into the portal layer the same way it came out. For further information on BONEs data structures and data types, please see the Designer Reference Materials.

Name	Type	Subrange	Default Value
DataSize	INTEGER	(-Infinity, +Infinity)	...
Tag	INTEGER	(-Infinity, +Infinity)	...
Source	INTEGER	(-Infinity, +Infinity)	...
Destination	INTEGER	(-Infinity, +Infinity)	...
Data_integer_vector	INT-VECTOR	(-Infinity, +Infinity)	...
Creation Time	REAL	(-Infinity, +Infinity)	...

Name	Type	Subrange	Default Value
Destination	INTEGER	(-Infinity, +Infinity)	...
Data	ROOT-OBJECT
Size	INTEGER	(0, +Infinity)	...
Source	INTEGER	(-Infinity, +Infinity)	...
Time of Creation	REAL	(-Infinity, +Infinity)	...

Figure A.13 – Portal Data Structures

The SCALE BONEs_MPI Portal has a much more flexible and logical interface. It consists of six ports: a send port, a receive port, and two sets of RTS/CTS (Return to Send/Clear to Send) connections. The BONEs_MPI portal will query the lower network layer if it is ready to receive a message with an RTS signal. If the layer is ready to receive, it should send a CTS signal back up to the portal layer. This protocol is non-polling so if the lower layer is not able to accept a message it must send a CTS signal to the portal whenever it is ready to receive (that is, if it received an RTS while it was busy).

Receiving data and sending into the portal layer is more flexible. It does work with the standard protocol (described above) but allows for a couple other alternatives. The RTS input can accept multiple outstanding transactions. In other words, the BONEs_MPI portal will remember how many times a request for a clear to send signal (RTS) arrived. Whenever there is buffer space available the portal will try to return outstanding RTS signals. Also, the user is not restrained by the protocol and has the option to ignore the RTS/CTS protocol and simply send incoming packets right into the portal layer. If the network model does not support reverse flow control then incoming packets might as well be streamed right into the portal queue, which is relatively large. However, as mentioned earlier with the older portal version, if the queue is filled, data will be dropped, eventually causing lockup. The data ports use the SCALE GMP data structure, which has five fields shown above in Figure A.13. The user must encapsulate this structure into a data field in the same manner described for the original portal version. The data size field in this structure is in bytes, and it must be extracted and used by the lower layers for proper network timing.

A.4. Currently Supported MPI Functions in ISE

MPI_Init
 MPI_Initialized
 MPI_Send
 MPI_Recv
 MPI_Barrier
 MPI_Barrier_time (not part of MPI standard)

MPI_Reduce

MPI_Allreduce

MPI_Bcast

MPI_Comm_rank

MPI_Comm_size

MPI_Get_processor_name

MPI_Get_count

MPI_Attr_get

MPI_Error_string

MPI_Error_class

MPI_Wtime

MPI_Wtick

MPI_Probe

MPI_IProbe

MPI_Finalize

MPI_Abort

MPI_Trigger_network - can dynamically change BONEs parameters (not part of MPI standard)

MPI_Force_delay - adds a specified number of seconds to the process's copy of the simulation time (not part of MPI standard)

No time calls - for profiling (Upshot) or for selectively placing communication overhead in program, a no time call for each function (not part of MPI standard)

A.5. Stages of ISE Development

Version 1.0 - Preprocessing (Static) Timing

Version 1.1 - HWIL, extended function list, Dynamic Timing

Version 2.0 - Simulated Processors (SWIL)

Version 1.2 - Distributed BONEs and Processes across Sockets, Upshot (HWIL)

Version 1.3 - Multiple Iterations

A.6. Other SCALE BONEs/MPI Notes

Broadcast - BONEs_MPI writes "-1" as a global address, for networks without broadcast capability this issue may be resolved with a block called "Bcast -> Mcast" that breaks the broadcast into separate sends.

Addressing Names - Nodes may be manually set to an address number, or BONEs includes the Address naming primitives which randomly assign a number to each node. BONEs/MPI

supports both and in either case, the rank is associated to the actual node address. If using BONEs Addresses see "Create Global Node List" below.

Create Global Node List - If using BONEs Address Names, a block called "Create Global Node List" must be in the system block diagram. This scheme is complicated and not suggested. It is also untested, so users may proceed at their own risk.

Other notes: the most practical of all MPI functions are supported in version 1.3 of BONEs/MPI. However, some key pieces are missing such as MPI Communicators. Also, as of this moment, there is some ambiguity of the functionality of *MPI_Bcast*. As far as we can tell; however, BONEs/MPI will follow the standard (and perhaps let the user get away with some tricks). Also for those who are intending to port their SCALE SHM model to BONEs/MPI, a conversion block has been built from "SCALE GMP -> SCALE SHM" for an SCI model; however, some adjustments may need to be made for other shared memory networks.

A.7. Sample Remote Host File for ISE

The following .rhost file shown in Figure A.3 has been provided as an example to demonstrate the format that should be used to take advantage of the ISE machine categorization. If the following format is not used, the ISE will simply not use multiple categories and will place all of the machines in a single category. Essentially, category names are designated in brackets immediately preceding the machines that fall in that category or type of machine. Note that this format should not affect the normal operation of the .rhost file with any other shells or programs.

```
[Sparc-10]
eagle markwell
eagle
[Sparc-5]
banshee
crusader
[Sparc-20]
blackbird
intruder
[Ultra-1 Reserved]
falcon
[Ultra-1]
cutlass
rapier
[Ultra-2]
dagger
dart
```

Figure A.14 – Sample .rhost File. This file format should be used to designate categories that the ISE can recognize to aid the user in choosing appropriate machines for process simulation.

Appendix B. Extensions to Parallel Conventional Beamforming

B.1. Extended Work in Time-Domain Beamforming

To extend the work completed on time-domain beamforming in the first year of this project, a number of optimizations were implemented. Two non-linear enhancements were added to the basic time-domain delay-and-sum beamformer in an attempt to bring its capabilities up to that of the FFT beamformers developed above. In addition, a new decomposition for the time-domain beamformer, the butterfly, was implemented and tested. Lastly, an algorithm written was written to sit on top of a Global Data Scope system.

B.1.1. Non-Linear Algorithmic Enhancements for Time-Domain Beamformers

The time-domain beamforming algorithm is of lower algorithmic complexity than comparable frequency domain algorithms. However, the amount of data that the time-domain algorithm requires is many times that of the frequency domain. Thus, in a parallel-distributed system such as the Fault Tolerant Distributed Parallel Sonar Array (DPSA), the communication cost overshadows the computational requirements. This section introduces two non-linear techniques to lower the data requirements for a time-domain beamformer.

The two non-linear techniques will be referred to as "Circular Shifting" and "Fractional Steering." The amount of computation is minimal for the Circular Shifting method and perhaps may even approach zero as will be explained in further detail. The Fractional Steering method is essentially a non-linear method of increasing spatial resolution without having to resort to interpolation techniques. It uses a well-known graphical interpolation technique introduced by Bresenham. The advantage of this technique is no increase in computation, which cannot be said for interpolated techniques.

The results of these methods are surprisingly accurate and offer tremendous savings to communication cost and simple data handling. Some spatial results are shown for conventional techniques in both domains and in comparison to the non-linear algorithms.

Conventional beamforming arrays are devices with a number of "dumb" nodes connected to an expensive front-end processor. This model is both inefficient and non-fault tolerant. It is inefficient in terms of energy and time, and the single front-end provides a single point of failure. With the advent of affordable microprocessors parallel computing is coming of age. In naturally (or embarrassingly) parallel systems, conventional computing techniques are illogical and impractical. One could analyze the slowdown in moving to a Von Neumann "sequential" paradigm. The DPSA is such a system.

The DPSA also has some physical limitations. The device must be autonomous, self-powered, and have an expected useful life of over 720 hours or 30 days. It must also be made up of small nodes powered by single C-cell sized batteries connected in a linear fashion. The power limitation is the greatest obstacle in creating the DPSA and limits both the computational power of the nodes and the communication network speed. Observing this, the DPSA must be outfitted with the most efficient hardware implementation along with an efficient beamform algorithm solution. This section addresses the latter for the conventional time-domain beamforming technique.

B.1.1.1. Circular Shift Method

The Circular Shift method is a means of reducing the time-domain beamform data space by drastically reducing the heights of sampled sequences. The time-domain data space increases as a function of $O(M^2)$ as opposed to the frequency domain which increases with $O(M)$ (recall, M = number of nodes). In other words, doubling the number of nodes on an array will double the data space of the frequency domain algorithm but quadruple the data space in the time domain. Also, doubling the resolution of the data space will increase the data space size by NM in the frequency domain and $NM + \frac{MS(M-1)}{2}$ in the time-domain (recall N = sequence size of one node). To explain this clearly we will revisit the conventional method of time domain beamforming.

The conventional time domain beamforming technique is simply a delay and sum of time sequences sampled by nodes in an array. In this case we will assume the array is linear equally spaced nodes. The delay and summing of these sequences is a means of amplifying or attenuating signals in a spatial conical field. The delay of the sequences determines at which angle the array is being "steered". Signals approaching at the steering angle will be amplified while others outside of the "beam" will be attenuated to some degree as shown in Figure B.1. A single sequence might look like the one shown in Figure B.2(a). If the spatial field is made up of a single monochromatic source, we expect to have a similar sequence recorded by all of the nodes but shifted in time, as shown in Figure B.2(b). The delay and sum of the data space matrix sweeps across a vector space of $N + \frac{S(M-1)}{2}$ (recall that S = number of steering directions)

which implies a total data space of $NM + \frac{MS(M-1)}{2}$. Notice, however, that it is possible to

have an extreme amount of redundant information if the signal is periodic. This is the fundamental theorem of the circular shift method: If signals are periodic (to some degree this can be approximate), the data space of the time-domain beamform algorithm may be reduced from of $NM + \frac{MS(M-1)}{2}$ to just NM . The data space is reduced in size to the frequency-domain

beamformer's, assuming the frequency domain algorithm is limiting the lowest frequency to maintain a complete cycle. The shifting of the time-domain sequence may now become a circular shift and thus approximate the true shift.

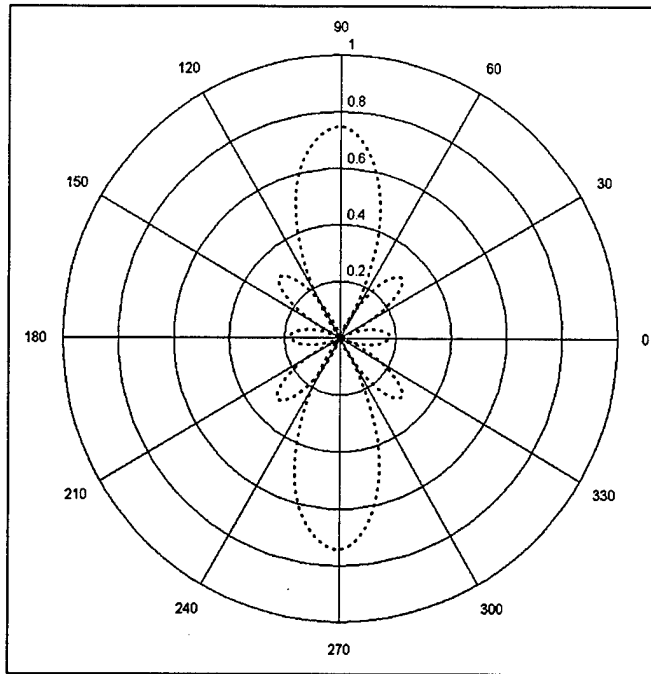


Figure B.1 – Beamformer Response versus Azimuth

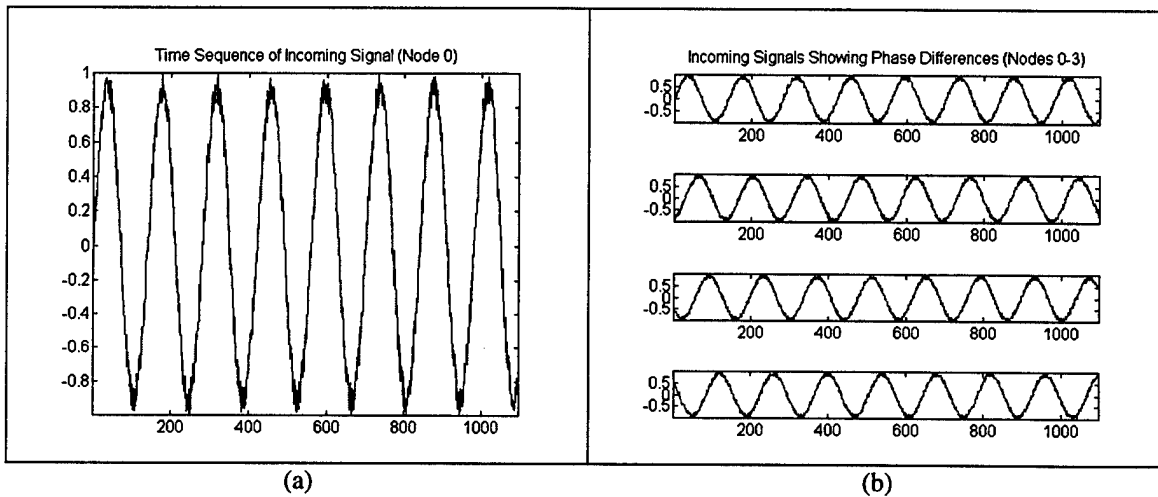


Figure B.2 – Typical Incoming Signals

This method requires two modifications of the conventional approach. The first modification is to create a circular shift of the time sequences. This is a simple (modification) improvement and contributes little added complexity to the system. The second modification may add a significant amount of complexity to the algorithm but not necessarily affect the execution time that much. For the circular shift method to work the sequence size N or periodicity of an incoming signal must be known. The complexity of this operation, due to an FFT and search, is approximately $N \log_2(N) + N$.

The method developed for the test programs may be thought of as an auto-beamform solution on one node's data set and is of higher order complexity than an FFT based solution. The procedure for "cycle_check" is developed further in the next sub-section. The alternative of this method is to perform an FFT on the incoming data samples and search for the highest peak and

approximate the number of samples in one period. Figure B.3(a) and Figure B.3(b) below show a sequence with redundant information and the resultant reduced signal after applying the “check_cycle” program, respectively.

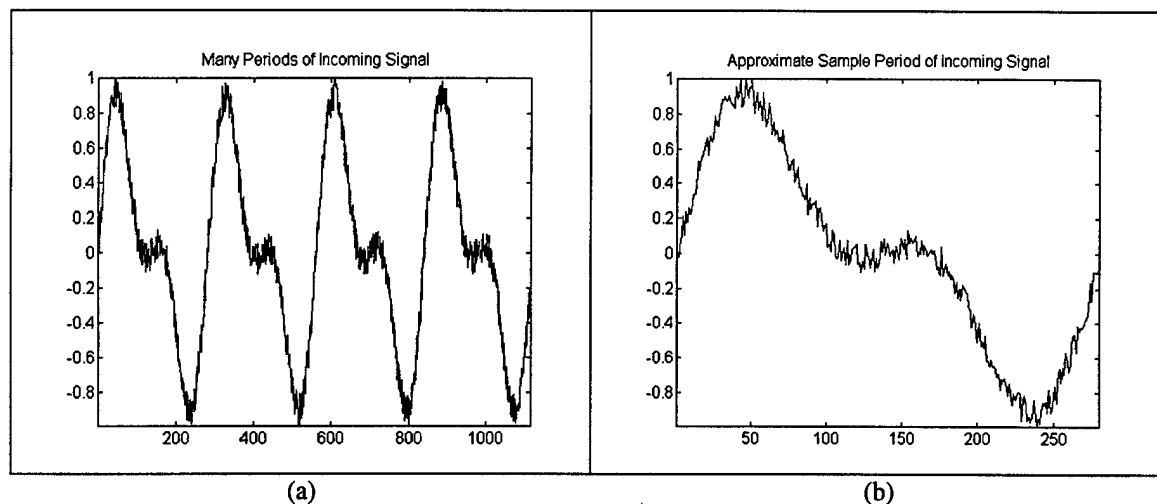


Figure B.3 – Incoming Signal and “Check_Cycle” Output

This method is surprisingly accurate (if signals are monochromatic) and in some cases introduces very little error to the conventional beamforming output. This is shown in Figure B.4 below. Introducing white noise (wide-sense stationary) into the system has little effect on the algorithm’s performance. We may also try more complex signals with multiple frequencies across the spatial field and the results still seem to be valid (Figure B.4).

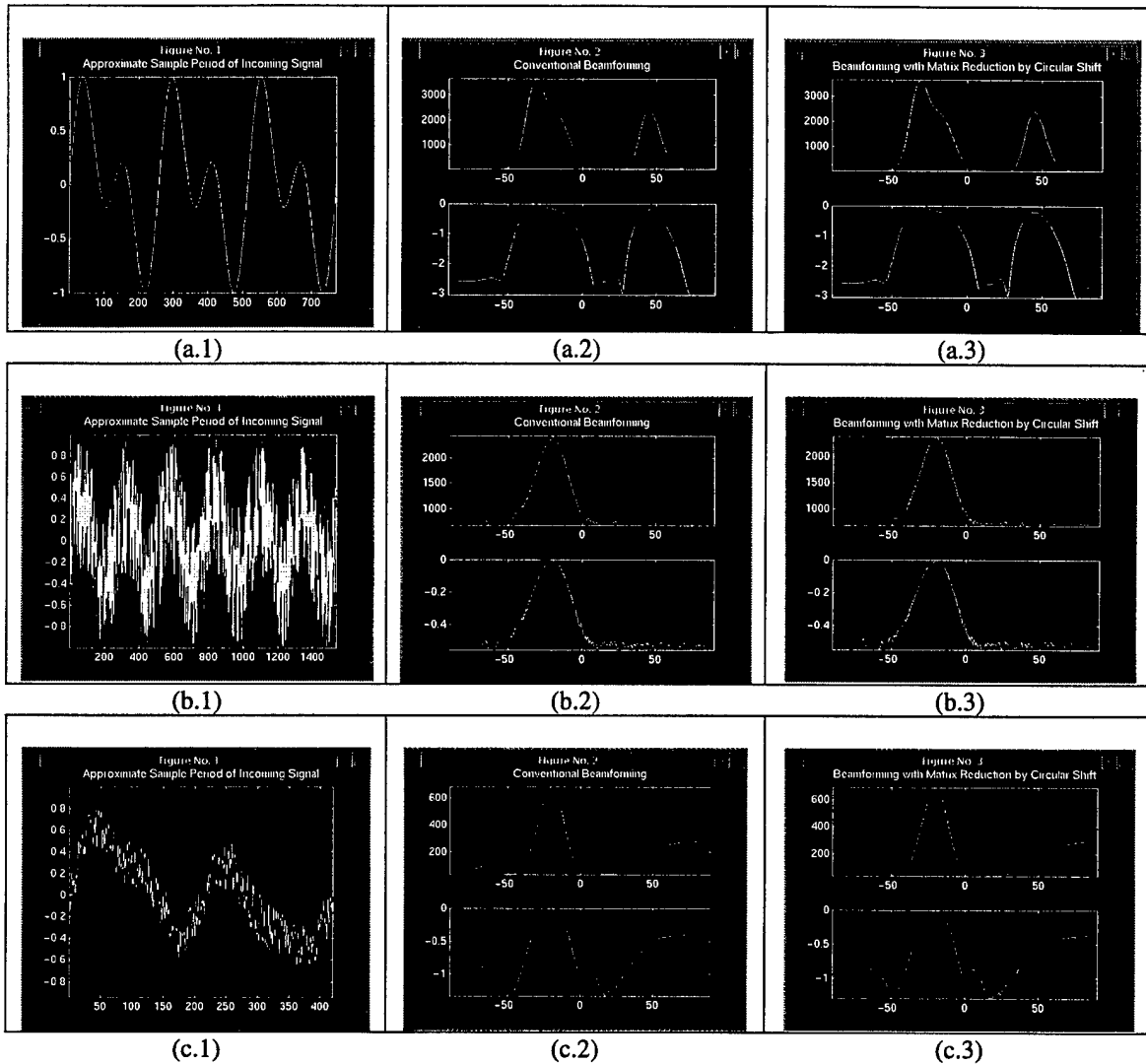


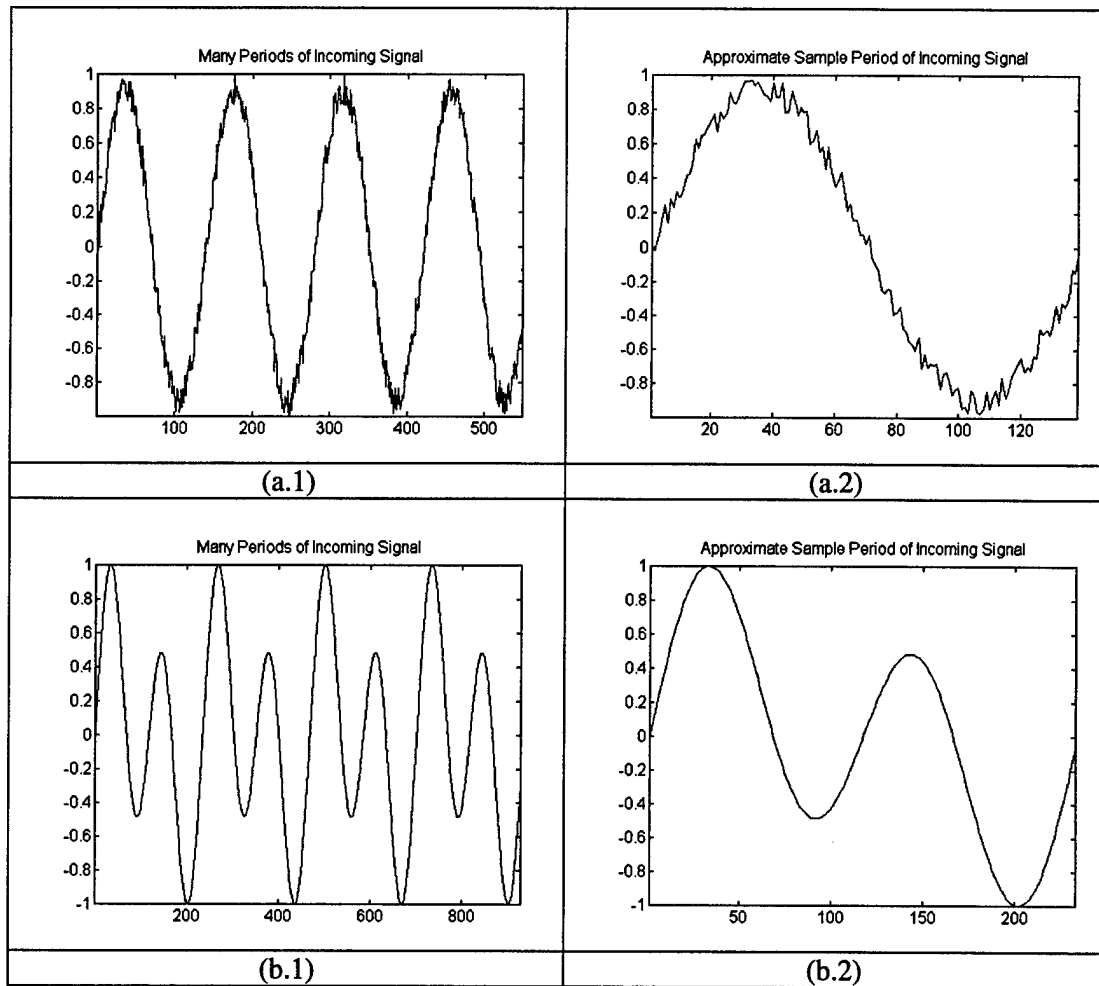
Figure B.4 – Circular Shift Method Results

B.1.1.2. Cycle Check Procedure for Circular Shifting

In order for the Circular Shift method to minimize error, a procedure may be periodically executed to acquire the approximate frequency of an incoming signal. Because incoming signals are likely to change slowly, the cycle test routine may be called infrequently saving execution cycles. However, it is likely to cause problems if the acoustic field is dynamic. A simple procedure was created to extract this information. The efficiency of this procedure was not optimized. In fact, this information could certainly be extracted with an FFT and search algorithm, which is of lower complexity than this method although perhaps less exact.

As described earlier, the method uses an auto-beamform technique. A cycle period is estimated (starting with the smallest cycle period possible of 2 cycles: the Nyquist frequency). The total number of cycles in the sample set are then added and the power of the resulting signal is found. If this power is greater than that of the previous cycle count, the new cycle count is recorded. A number of signals were applied to this algorithm to test its validity. Signals were

applied which had multiple frequencies (and thus had a complex cycle), such as broadband signals and signals with frequencies that were not integrally related to the sampling. The outputs in Figure B.5, below, show many cycles of the input signal (left plots) and the resulting signal when truncating to its approximate cycle count (right plots). The first three cases perform very well and find the cycle count to within an error of a couple of samples. However, the last case, which is made up of three signals at 36, 72, and 90 Hz, breaks down, finding the half-frequency instead of the true full cycle. However, applying cases like these into the beamform algorithm do not seem to make a difference and results still seem to exhibit little error.



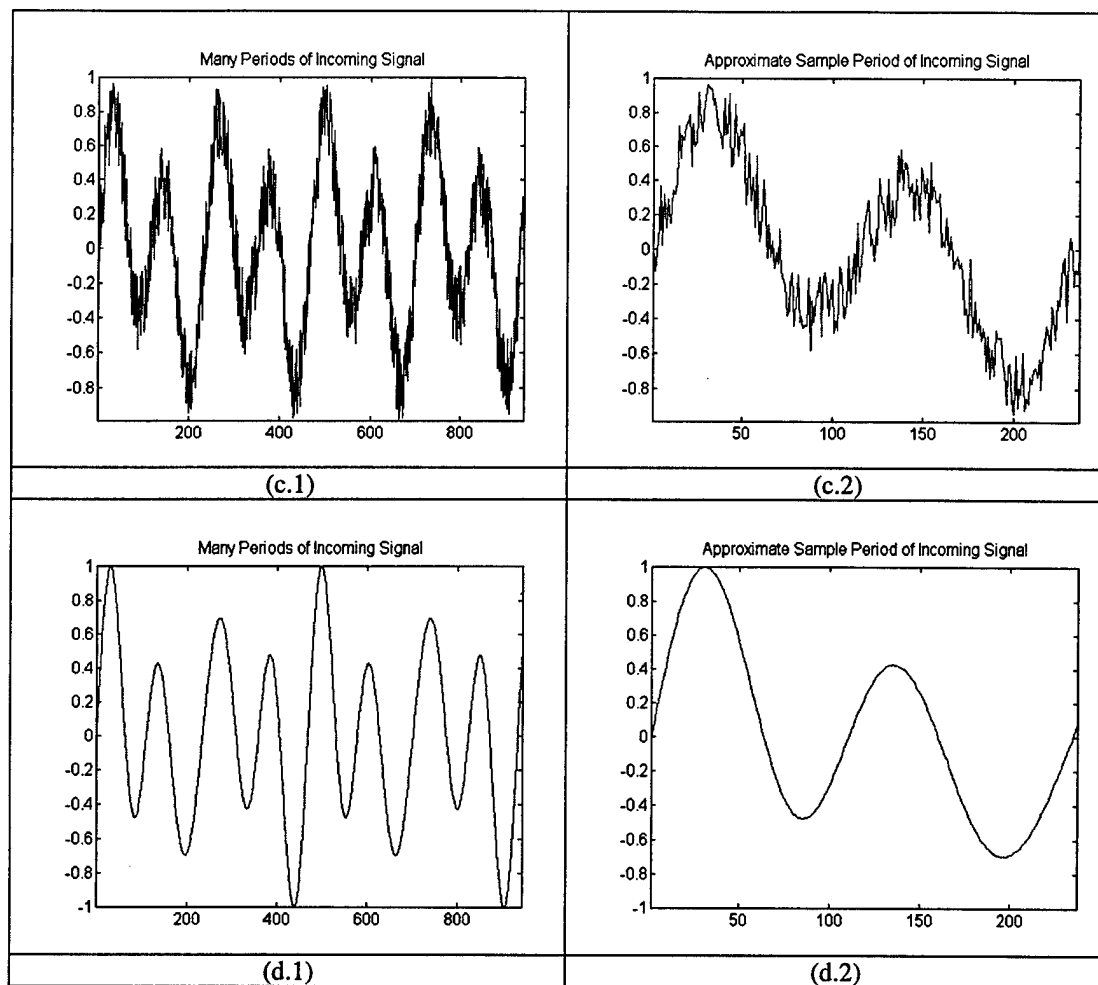


Figure B.5 – Experiments with the “check_cycle” routine.

B.1.1.3. Fractional Beamsteering

Fractional Beamsteering is an alternative of time-domain beamforming with interpolation. It is based on Bresenham’s algorithm (described below) for drawing lines on graphic displays. By fractionally steering across the data matrix, better resolution can be achieved at the cost of increased distortion. However, the distortion is a function of the number of nodes in the array. More nodes imply less distortion. This is convenient since large arrays will have a bigger communication and data-handling problem. Fractional Beamsteering has a limit on the number of fractional directions that can be achieved. The maximum number of fractional resolution angles is equal to $M - 1$; therefore, an eight-node array can achieve a fractional resolution of seven times its natural capacity, and a sixty-four-node array can achieve sixty-three times its natural capacity. Of course, the larger the number of fractional angles, the more distortion there will be in the final signal and the more computation is required. However, this is a better alternative to interpolation with large arrays since it achieves the same result: a pseudo-increase in angular resolution. This is because interpolation requires a large amount of computation whereas Fractional Beamsteering requires considerably less.

The method utilizes Bresenham's algorithm to do non-linear shifts across the data matrix. Notice in Figure B.6(a) that in conventional beamforming, the summation reduction across the data matrix requires node 2 to step in incremental amounts. This requirement forces nodes further down the array to jump with great hops if the number of steering directions is sufficiently large, and the last node down the chain to shift at $S(M-1)$ wide (where S is the steering angle offset). For a 64-node array, this implies that the 64th node will have to jump down to its 63rd sample when beamsteering one increment to the left. At two increments to the left, the 64th node must start the beamforming process 126 samples down. This is easily shown to blow up when the resolution required is very high. The data space becomes very large. Fractional Beamsteering, with greatest resolution, does not require node 2 to step in incremental amounts, but instead requires the furthest node down the array to jump in incremental steps. Thus, the line of incidence is interpolated by Bresenham's algorithm, again at the cost of some distortion. However, this distortion may be minimized in large arrays by requiring the furthest node to make a longer hop. For instance, in a 64-node array the hop increment can be set to 7 thus increasing the resolution by eight while minimizing distortion.

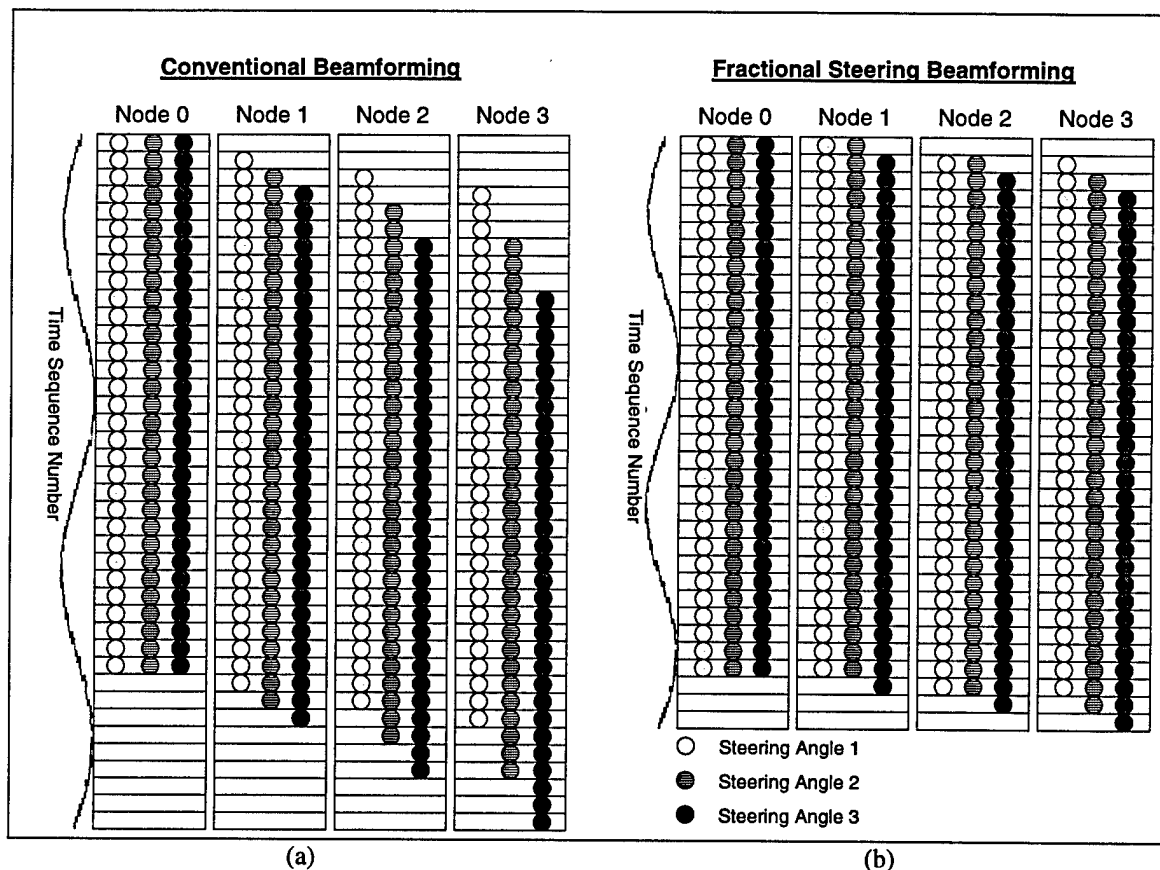


Figure B.6 – Fractional Steering

The results of this method are surprisingly accurate even for incremental hops at the furthest node in the chain and for small networks. Figure B.7(a)-(c) below shows the plots of the Fractional Steering method and Fractional Steering and Circular Shift together for the same inputs as shown in Figure B.7(a)-(c). Notice that the distortion is much greater; however, the actual system will perform better since it will have many more nodes. In this case there were only eight nodes and the maximum data compression was applied by reducing the amount of data

by seven. Also note that because there is less data the signal seems to have less dB gain over the noise. Distortion and lower signal gains are two drawbacks to the Fractional Steering method.

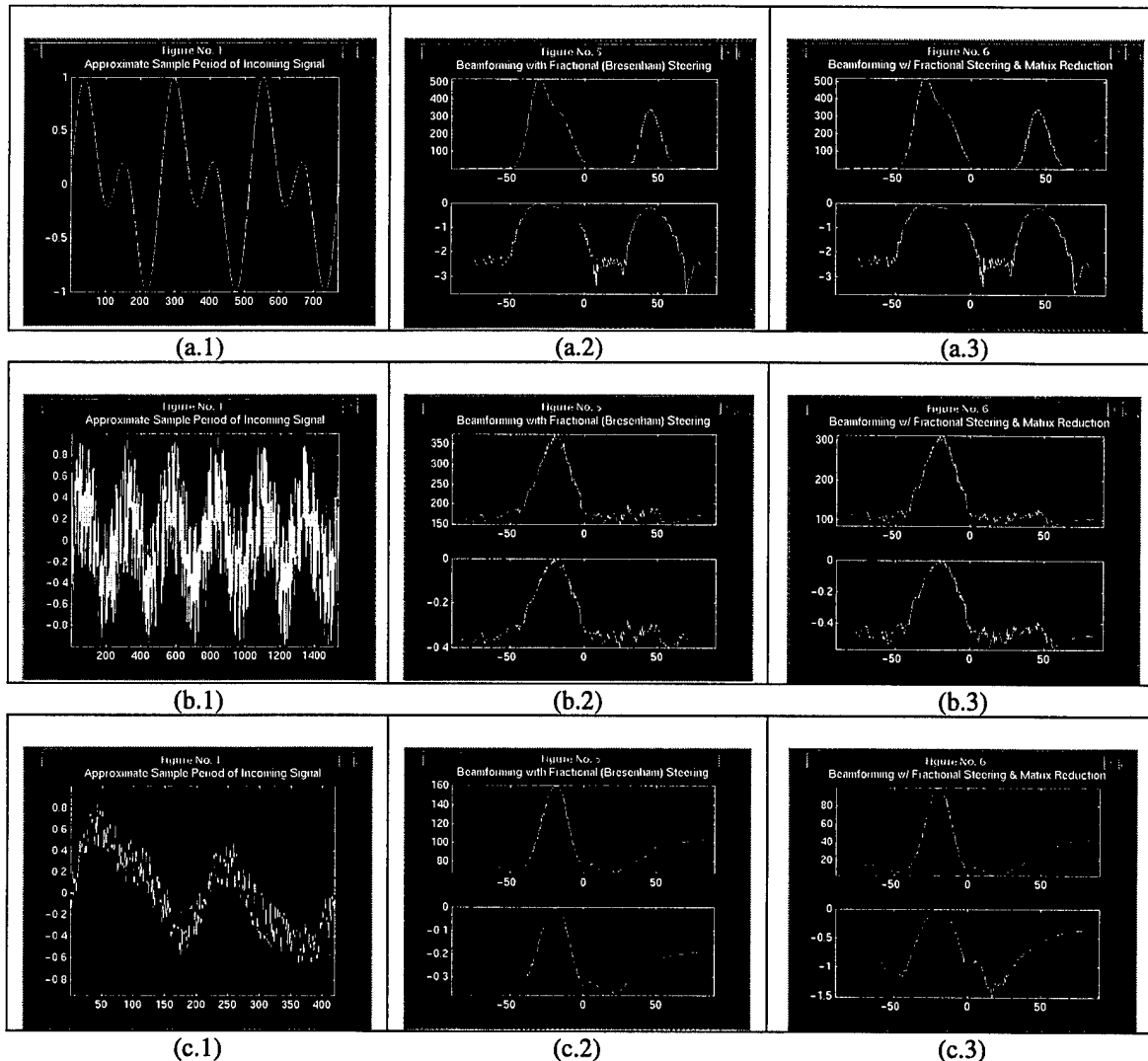


Figure B.7 – Fraction Steering and Fractional Steering with Circular Shift Results

Using both the circular shifting method and the fractional steering in concert the beamforming system saves a great deal in communication and computation cost, so much so that this algorithm may become a better alternative than the FFT beamforming technique. This is especially true if the target system requires low communication and computation.

B.1.1.4. Bresenham's Algorithm for Fractional Beamsteering

Bresenham's algorithm was invented to approximate or interpolate lines across a graphic display matrix. Since displays have finite resolution, often the lines must be interpolated. The technique is simple but requires some explanation. First, the axis of control, the "driving axis," is determined. The driving axis is defined as the axis for the direction in which the desired line has the longest dimension. In Figure B.8 below, the driving axis is the x-axis (horizontal). The other axis may be referred to as the "passive axis", in this case, the y-axis (vertical). A pixel is illuminated if a line crosses its passive axis on the side nearest to the origin of the line. Notice,

then, that direction of the line is arbitrary but yields two different solutions. For instance, the example figure below shows that the algorithm was applied in the direction from (0,0) to (5,3). If, however, the example was applied from the opposite direction, (5,3) to (0,0), then pixels 15, 14, 8, 7, and 1 would be illuminated. Careful observation shows that this is the exact same line if turned 180°.

The method uses an error function ϵ to determine if the true line has passed through the pixel at the same passive axis coordinate or through the pixel above the current passive axis coordinate. First the slope m is calculated by conventional algebraic conventions. The slope is added to the error variable ϵ , and if the resulting value is greater than 1 the passive axis is incremented, and 1 is then subtracted from the error ϵ and the process continues.

This algorithm can also be solved for non-integral starting and ending positions. However, for the time-domain beamformer, samples will always be integrals. Therefore, the simple integral case need only be considered. To maximize the efficiency of this technique the interpolated lines may be pre-calculated and stored in a matrix.

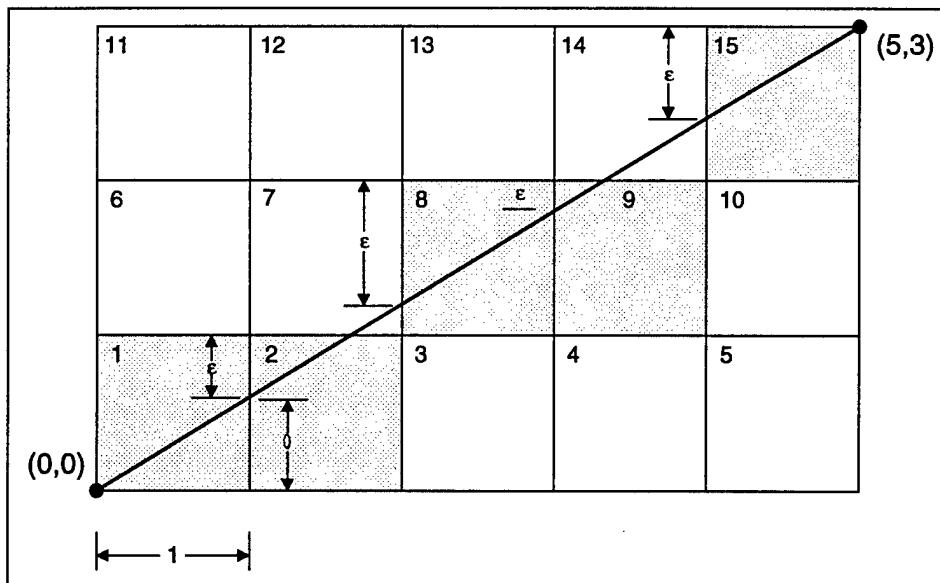


Figure B.8 – Interpolated Line of Bresenham's Algorithm [JOY97]

B.1.2. The Butterfly Beamformer

One beamformer of interest is a fine-grained butterfly algorithm. The technique is called butterfly because it has the same characteristic as the Fast Fourier Transform. This algorithm is more fine-grained than other previously implemented beamformers such as PBT (Parallel Bidirectional Time-Domain), a pipelined control parallel approach or GDS (Global Data Scope), or a coarse-grained agenda parallel approach. The algorithm is optimal for low-latency computations, however, its speedup is limited by the complexity of the operation.

B.1.2.1. The Butterfly Characteristic

The butterfly is so named because of the characteristic of its main computation a data parallel matrix reduction. The reduction is the summing of shifted time sequences into a resultant vector.

This vector then may be reduced to a power by taking the square of its terms. The Butterfly technique then operates in two stages: the matrix reduction, then vector reduction. The matrix reduction actually performs the reduction of two steering directions simultaneously and for simplicity finds the complement angles, leaving the broadside-steering direction as a special case. This process must be performed in a loop until all of the steering directions are complete. Figure B.9 below shows a flowchart for the butterfly matrix reduction. The odd nodes perform the matrix reduction of a positive steering beam while the even nodes perform the reduction for the complement negative direction. The first stage of this operation is the swapping of data between odd and even nodes. After this stage the odds and evens are completely independent. The algorithm then follows a recursive structure in which each node sums two vectors, splits the resultant vector into two half vectors, keeps one half vector and swaps the other with its complement node. The complement node algebraically is difficult to express but is easily seen in Figure B.9 as the node to which another is sending for that stage. The operation of adding, halving, and swapping continues until each node has added its component to a resultant distributed vector. Note that the data vectors for each node must start as radix 2. The next stage in this algorithm is to collect both of the distributed vectors and possibly calculate a power for each. Unfortunately, this stage is a sequential bottleneck which could only be overcome by a complicated pipelining (control parallel) technique. After the power is performed for each steering angle the algorithm loops back to stage 1, looping approximately half the number of steering angles required of the system. Figure B.10 shows the high-level diagram of this delay-and-sum algorithm.

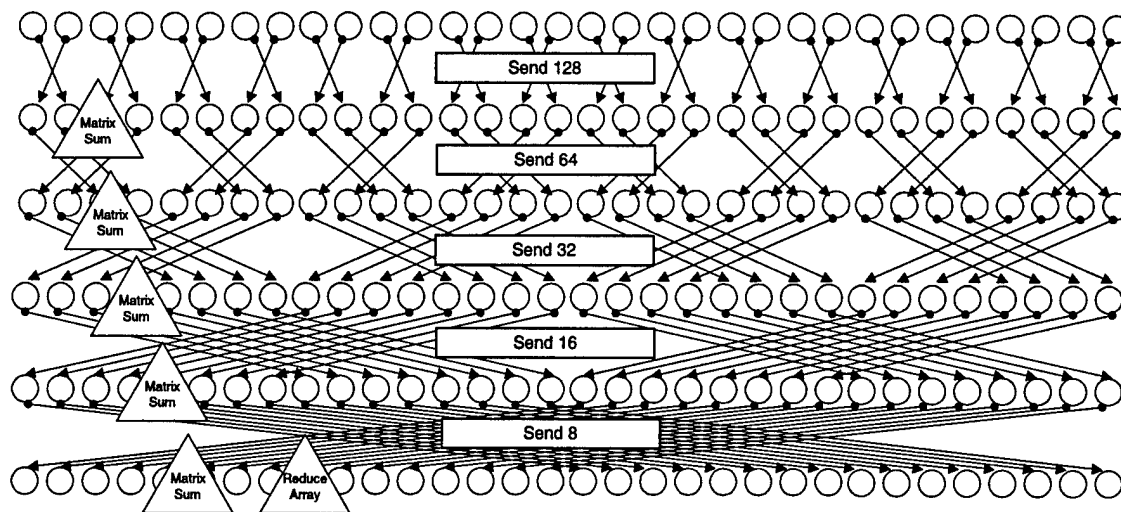


Figure B.9 – Butterfly Detailed Flowchart

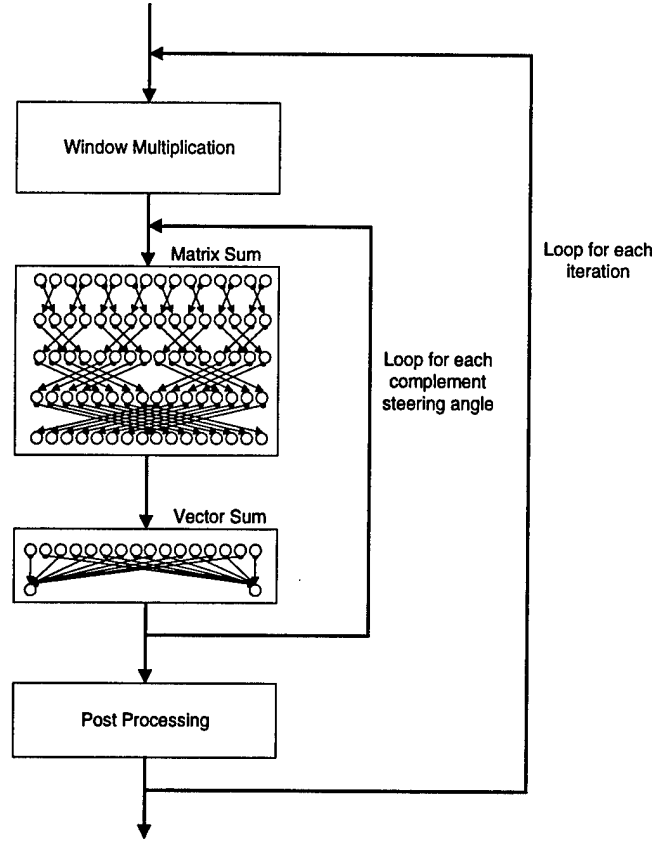


Figure B.10 – Butterfly High-Level Flowchart

B.1.2.2. Complexity of the Butterfly Beamformer

The Butterfly Beamformer has the limitation of being too fine-grained. Because of this limitation, the communication cost becomes a degrading factor to its performance. Analyzing the stages of the Butterfly the complexity of the algorithm may be mapped with the following equation:

$$\text{Ops}_{\text{butterfly_conventional}}(S, N, M) := \frac{t_{\text{matrix_sum}}(S, N, M) + t_{\text{window_mult}}(M, N)}{M} + t_{\text{find_energy}}(S, N) + t_{\text{pick_angle}}(S)$$

Recall M = Number of Nodes, N = Number in Time Sequence, and S = Number of Steering Directions.

This equation is explained in further detail in the section on performance prediction. It is easy to pick out the sequential bottleneck terms of this method. The beamformer is optimized for low-latency but not high throughput, whereas the pipelined beamformer is optimized for high-throughput but arbitrary latency. Lastly, the GDS time-domain beamformer system (discussed next and in the performance prediction section) is superior in both regards offering both lower latency and high throughput. Comparing the Butterfly to the other two time-domain beamformers shows that it is of higher order complexity. Worse yet is its inability to take advantage of data

compression with “Circular Shifting” methods (discussed in the previous section). This leads to very high communication cost. The communication pattern may be written as:

$$\text{Comm_ring_butterfly_conventional}(M, N, S) := t_{\text{comm}}(S \cdot M \cdot N, M, \varepsilon, \psi_{\text{ring}}(M)) + t_{\text{comm}}(S \cdot N, M, \varepsilon, \psi_{\text{ring}}(M))$$

The communication pattern is broken up into the two stages similarly to the computational equations. The first stage is the communication for the butterfly matrix reduction and the second stage is in collecting the distributed resultant vector to a node or nodes. The communication complexity is far greater than the pipelined or GDS techniques because there is a tremendous amount of redundancy of the data for each steering loop. It is likely that storing samples across steering angles may minimize this communication, however, this adds even further computational complexity to the system. Besides being analytically studied as higher order complexity, the algorithm presents itself as highly irregular and is likely to add further complexity when programmed. Even worse is its limiting fault-tolerance. For the butterfly to compensate for a failed node would be a programming nightmare and would add considerable latency to the system.

B.1.2.3. The Delicate Butterfly

Although interesting as a beamform paradigm, the butterfly beamformer is an unlikely candidate for the DPSA. The system is simply too fine-grained to be competitive with the alternatives. Besides being the algorithm with the highest algorithmic complexity, it undoubtedly has the worst fault-tolerance, making it a delicate system. It does have several advantages, however. It requires the least amount of memory and theoretically can perform a single steering angle decomposition better than any other method implemented.

B.1.2.4. A Global Data Scope Time-Domain Beamform Algorithm (GDS-TDBA)

The Global Data Scope Time-Domain Beamform Algorithm (GDS-TDBA) project is a new and important dimension in the study of parallel beamform for the DPSA. Although GDS-TDBA itself is a beamform algorithm, it more generically creates a simulation environment to observe the subtleties of a real-time environment. The DPSA is an autonomous sonar system that must continuously sample the environment and reduce this information to a meaningful picture of the sample space. Operating in such a mode requires that the algorithm pay particular care to the nature of the system. That is, the algorithm must operate on a set of boundaries which dynamically change. In addition, the algorithm and the sampling parameters will dictate the size and performance of certain devices in the system such as memory and the interconnection network.

The real-time beamform array could be compared to a dataflow architecture in which the array is silent until data begins to move into the system. This is in contrast to the von Neumann style programming typical in modern day computers. In systems of this nature, data resides in a static space and are transformed when pulled into the algorithm. Dataflow is more natural for a system in which the data pushes itself into a data space, and worse yet, is likely to write over other data at some determinate time span. In the case of the DPSA, an A/D converter is likely to be such a device with the possible aid of a DMA (Direct Memory Access) controller.

The GDS-TDBA assumes such a system and investigates some of the non-trivialities of operating an algorithm in such an environment. Some of the assumptions of this system include

the ability to occasionally drop a sample frame, ADCs which are synchronized within some tolerance, the ability for the algorithm to ignore synchronization problems, and the inability to operate the architecture in a collect and wait protocol. Each of these assumptions will be discussed in further detail below.

B.1.2.5. The GDS-TDBA Algorithm

The GDS-TDBA is essentially a time-domain beamform algorithm that assumes a fully functional network. GDS-TDBA is meant to be an alternative to Parallel Unidirectional Time-Domain Algorithm (PUT). PUT was, in contrast, a fine-grained beamformer. However, both share a common pipelined philosophy. That is, both algorithms attempt to gain speedup by hiding the latency in sending signals on the network. GDS-TDBA is not a performance study on beamforming algorithms, but an algorithm to study the complexities in creating a real time system capable of continually sampling and beamforming. This philosophy of sampling and beamforming continuously, is contrary to the alternative of a sample, stop, beamform, and sample continue protocol.

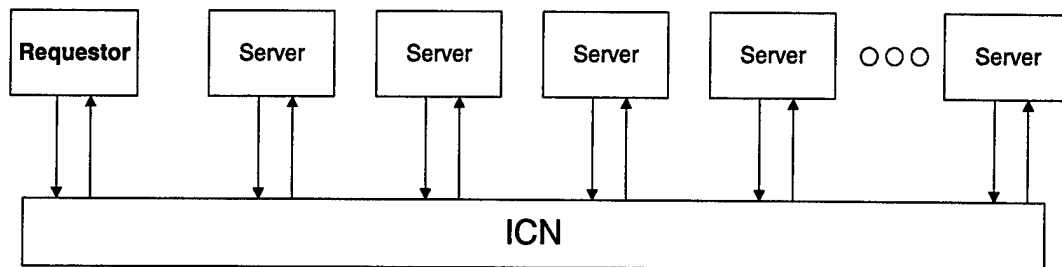


Figure B.11 - Requestor/Server Paradigm

The requestor is in charge of doling out jobs to the servers. In a fault-tolerant system a server can become the requestor if the requestor were to fail.

GDS-TDBA uses a requestor server paradigm in which the requestor is stationary. Its job is to dole out jobs to the servers, collect and sort the beamform data, and perform any post-processing tasks. Although, it is stationary, fault-tolerance is not reduced if other nodes have the capability to perform the requestor's duties if it should fail. A more complicated scheme could use a roaming requestor, in which case the requestor is not really used to dole out jobs but rather to compile the beamform data. Figure B.11 above shows the generic structure of such a system.

GDS-TDBA is a low-latency algorithm due to latency hiding. In most beamform algorithms the processing node needs to first collect samples from an array of nodes through some interconnection network (ICN). GDS-TDBA assumes that the network can continuously feed data from all of the other nodes in real time without overloading the network mechanism. Therefore, when a node is asked to produce a beamform result, the data is already present in the node and reduction of a matrix field can occur immediately. In other algorithms the node must wait large periods of time while samples are gathered from its local analog to digital converter (ADC) and other nodes. The feasibility of placing this much traffic may be analytically estimated with the following:

$$traffic = 32 \text{ nodes} \cdot \frac{31 \text{ destination packets}}{\text{nodes}} \cdot \frac{64 \text{ bits}}{\text{packet}} \cdot \frac{1000 \text{ times}}{\text{second}} = 64 \text{ Mbps}$$

Equation B.1

This equation assumes a 32 node array, a 64 bit packet (including package type, source, destination, sample, iteration, and routing/network overhead), and a sample rate of 1000 Hz. A 64Mbps network is feasible but not under the low power constraints of the sonar array. However, using a broadcast send would lower this traffic to 2 Mbps, which is certainly feasible. More importantly, the broadcast network is scalable as well for larger arrays.

B.1.2.6. GDS-TDBA Memory Matrix Solution

GDS-TDBA requires that all of the nodes operate in a synchronized fashion with moderate precision. Assuming this, the protocol for gathering data is simple. Each node holds a large matrix and an array of pointers to this matrix. The matrix is 2-dimensional with each column belonging to a node number and each row holding the most recent transducer sample. The pointer field array holds a pointer for each column or node. When a sample is converted locally or a sample is received from the network it uses the pointer to find its destination in the matrix. Since there is undoubtedly some time delays between each sample, the data fills the matrix in a sweeping fashion much like sprinters on a home stretch. Although the local node is sure to win each race, the samples when properly synchronized should fill the arrays at approximately the same rate. In other words, at most one column might be one sample ahead of another. This is a safe assumption considering the low sampling frequency of the sonar array. Eventually, the samples will get to the end of the array and must then wrap around to the beginning or first row. One problem in this type of system is that samples can overwrite data, which may be in the middle of a reduction operation.

When asked to perform a beamform reduction operation each node must simply locate one of the pointers of the matrix field and begin the beamforming operation at that point. Memory size is a critical concern for this type of operation and is dependent on a number of factors, some of which are known, many of which are not. Using some of the assumptions aforementioned (such as synchronization), the size of memory (actually the size of each data column) may be estimated with the following:

$$length = \left\lceil \frac{\text{steering directions}}{2} \cdot (\text{number of nodes} - 1) \right\rceil + \text{normalized beamform time} + \text{setsize}$$

$$\text{where, normalized beamform time} = \frac{\text{time of beamform computation}}{\text{sampling period}}$$

Equations B.2, B.3

The *setsize* variable may be on the order of 64 to 128 samples. The number of steering directions may be as few as 9, and as many as 181. The normalized beamform time is the least known variable. This is due to the fact that currently the processor speed, the computational requirements of the beamform algorithm, the sampling period, the overhead of the network protocol, etc. will all have a significant effect on the normalized beamform time. In addition, the algorithm could prioritize the samples that are likely to be overwritten first and thus minimize the matrix size. A rough estimate of the size of the memory was used to create the matrix for the GDS-TDBA simulator. Assuming 180 steering directions, 32 nodes, a set size of 128 samples, and a normalized beamform time equal to the set size, the total column length would have to be

3046 samples. The total sample memory space for a node may be a 32 by 3046 matrix. Of course, the node requires additional memory for computational requirements. Note that this memory calculation was based on the assumption of a conventional time-domain beamform algorithm. Other algorithms, especially the frequency domain beamformer or time-domain with circular shifting, can reduce the size of the memory by the factor $\frac{1}{2}S(M-1)$, where S is the number of steering directions and M is the number of nodes.

B.1.2.7. GDS-TDBA Parallelism

The beamform algorithm used in this section was a simple time-domain conventional delay and sum. The delays are used to beam steer and the data is reduced into a single array which is then reduced again into a sum of squares or the beamform power. Figure B.12 below shows this operation. Parallelism of the array is achieved by each of the nodes operating on the same data but performing different steering directions. Beamsteering is created by a node reducing the matrix of shifted sequences. Figure B.13 shows two matrices reduced with a degree of 1 and a *setsize* of 8. A node may either steer in a positive direction or a negative direction. If steering in a positive direction the left most (or largest node number) array pointer is used as a basis for the matrix reduction. Note that time is flowing down the page so steering positively requires the furthest sample in time to be at the left most array, this conversely becomes the starting point. The opposite is true when steering negatively. The steering directions could be doled to the "servers" from a requestor or each node could be in charge of a certain beamforming direction, which is hardwired into their logic. The latter would simply reduce the communication requirements but lessen the fault-tolerance. In either scenario a requestor or master is required to collect and sort the beamform powers. The requestor, by finding the node that sent the maximum power, could estimate the source of an oncoming signal. Note that in a fault-tolerant system the requestor in a requestor/server paradigm as shown above in Figure B.11 could be replaced if it were to fail by any of the other server nodes. This functionality was built into GDS-TDBA by placing the requestor's address on every request to each of the servers. The servers will always send to this most recent address. Also, the requestor just like every other node must also contain a transducer and send samples to each of the other nodes. As mentioned previously, the interconnection network is assumed to have full connectivity to each of the other nodes and must be able to broadcast messages to lessen communication requirements.

The GDS-TDBA from an analytical viewpoint seems to have the best characteristic of all of the time-domain beamform algorithms. Since data is constantly fed to each of the nodes prior to the beamform request, the second the request arrives, each node responds following a prescribed agenda (which steering directions to complete). Analytically, the pipelined PBT algorithm and GDS-TDBA are identical. From a programming standpoint, though, the GDS system is simpler to implement and therefore, likely to be the superior algorithm. It should be noted at this point, however, that GDS is not limited to the time-domain conventional beamformers. Any technique may be employed such as FFT conventional, Matched-Field, etc.

Beamform Operations

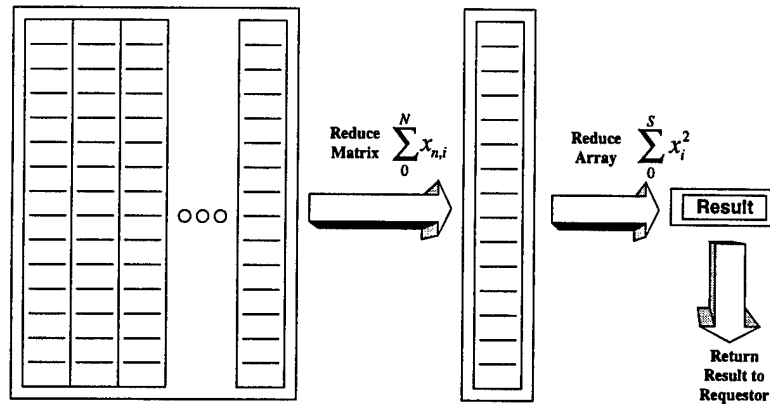


Figure B.12 - Beamform Reduction Process

The beamform reduction operation occurs in two stages. The first stage reduces a large matrix into an array of summations. The second stage reduces this array into a sum of squares divided by the array size or power.

Matrix Reduction

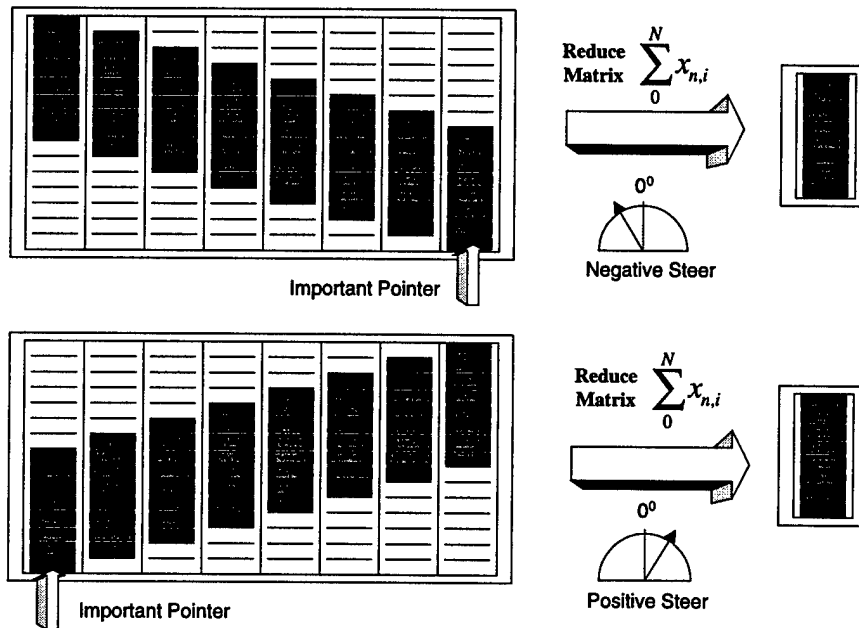


Figure B.13 - Beam Steering and the Reduction Operation

The reduction of each node's matrix depends on the steering direction. A positive steer requires the left most pointer as the base address, and a negative steer requires the right most pointer. When a node is asked to beamform, it immediately gets one of these two pointers to base the reduction operation.

B.1.2.8. GDS-TDBA Source Code

The GDS-TDBA simulator was created using MPI standard extensions to the C language. The program works in a quasi-data flow type system. The data movement controls the tempo of execution. Both the requestor and servers operate in an infinite loop, and periodically update their local memories from a sample file. This sample file could have been written into memory as one large block but again the intention of this project was to attempt to create a real-time system in which samples are written into memory at a set rate. The rate at which this data is written into each node's memory depends on a couple of factors. The first is the sample rate. To lessen the chance of error the user is not able to set the sampling rate of the GDS-TDBA program. Instead, this rate is read off of the sample files to ensure that both of them are operating under the same assumptions. In addition the speed of sound and the distance between nodes is also written into the first lines of each sample file for the nodes to pull off. The second factor that affects the rate at which data is written into the program is the *CLOCK_RESOLUTION* factor. A local clock holds the current clock time and a sample is written into memory after the sample period is equal to the clock counter minus the last sample time. After the sample is written into local memory, each node also broadcasts the data sample to each of the other nodes. In the case of the GDS-TDBA simulator the *MPI_Bcast* was not used but rather each node sent a direct message to all of the other nodes. This was due to the limitation of the *MPI_Iprobe* routine, which is a non-blocking test for receives and does not work with the broadcast function. At every iteration nodes also check for incoming data and incoming requests. The *MPI_Iprobe* is placed in a loop and checks to see if data is coming in of type *SAMPLE_PAK*. A sample pack holds the latest sample value and an iteration number. This iteration number is a simple form of insurance to ensure that samples are somewhat aligned when beamforming. *MPI_Iprobe* is also used to search for messages of type *REQUEST*. However, an *MPI_ANY_SOURCE* flag is used since the requestor may occupy any address and may change at any time.

Each node may service a number of different requests: reinitialize matrix memory, beamform, or closedown. The last request is simply a tool for the GDS-TDBA simulator and would not be present on the actual parallel sonar array. One note about servicing requests, a node may not service a new request if it is the middle of a beamform reduction operation. It must wait until it is complete then service the new requests.

Careful analysis was paid to the intricacies of creating a real-time environment in which to place a beamform algorithm. Parallelism of the beamforming result could be achieved through many techniques such as pipelining or result parallelism. The particular approach lends well to a result parallel approach in which every node operates on the same data space but extracts different information from it, in this case different beam steers. A pipelining approach could overlap execution of a full beamform solution to increase the throughput of beamform solutions. However, this would likely increase the size of memory.

Future work will require careful inspection of the least known variable, normalized beamform time. Through careful modeling of a projected processor and statistics of real beamform solutions, whether that be adaptive, matched-field, etc., this parameter will become known and further suggest the boundaries for other devices such as memory size, computational requirements, modes of operation, etc.

B.2. Extended Work In FFT Conventional Beamforming

This section presents work pursued in this project to augment the core results presented above. First, a new medium-grain alternative is presented. The new medium-grain full-capability FFT beamformer extends the version presented above by creating a pipeline of communication

stages. These two programs will be referred to as the non-pipelined program and the pipelined program. Rather than sending data to begin a communication stage, then receiving it in the same iteration to continue the computation, the new alternative begins a communication stage by sending data to a future iteration and finishes the communication stage by receiving from the previous iteration. Second, the FFT conventional beamformer programs are changed to allow measurement of individual sections of the program. The algorithms are split into categories based on the beamforming functions performed and the categories are analyzed individually. The execution timings of the individual categories within the beamforming programs allows a better understanding of where exactly the computational requirements are higher and where parallelism can be best utilized. This section includes a description of how the categories are defined and presents the measured execution times by category for the existing parallel programs.

B.2.1. Pipelined Medium-Grain Full-Capability Algorithm

One of the most important limitations of the original medium-grain full-capability parallel FFT beamformer is the large amount of communication. Because every node must have the full data matrix to compute any assigned steering directions, the communication is a much more important factor in execution time than is the case for the coarse-grain algorithm. A new implementation of this algorithm in which the communication costs are reduced is presented here.

Figure B.14 shows the flowchart for the original algorithm. The FFT and shading factor multiplications are followed by a communication stage in which all nodes receive the full data matrix. After this, each node calculates its assigned steering directions then sends the results to a single collection node. The pipelined algorithm improves upon this procedure by making pipeline stages out of the computational blocks. The communication between these blocks no longer occurs within the single iteration. Instead, each communication is comprised of sending to a future iteration and receiving from the previous iteration. When the nodes have completed the FFT and shading factor multiplication for Iteration 3, they send out their data columns to the other nodes. Instead of receiving these data columns (which would require blocking until the data has propagated through the network), the nodes receive the full data matrix from the columns sent during Iteration 2 (which has already propagated through the network during the previous iteration's execution time and has been buffered). Then, the nodes calculate the assigned steering directions based on the data from Iteration 2. When done, the nodes send the results to the front-end node for collection. At this point the front-end node does not block to wait for these results. Instead, it receives the results for Iteration 1, taking advantage of the fact that those results have had enough time to propagate. Figure B.15 shows the flowchart for this algorithm. The solid lines and boxes represent the flow of a single data set through the three-stage pipeline. The shaded lines and boxes show the algorithm operations for other iterations.

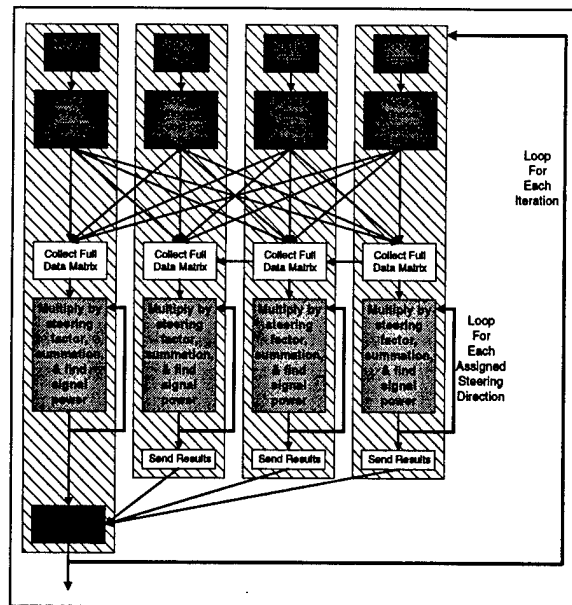


Figure B.14 - Original Medium-grain Full-capability Network-independent FFT Beamformer Flowchart

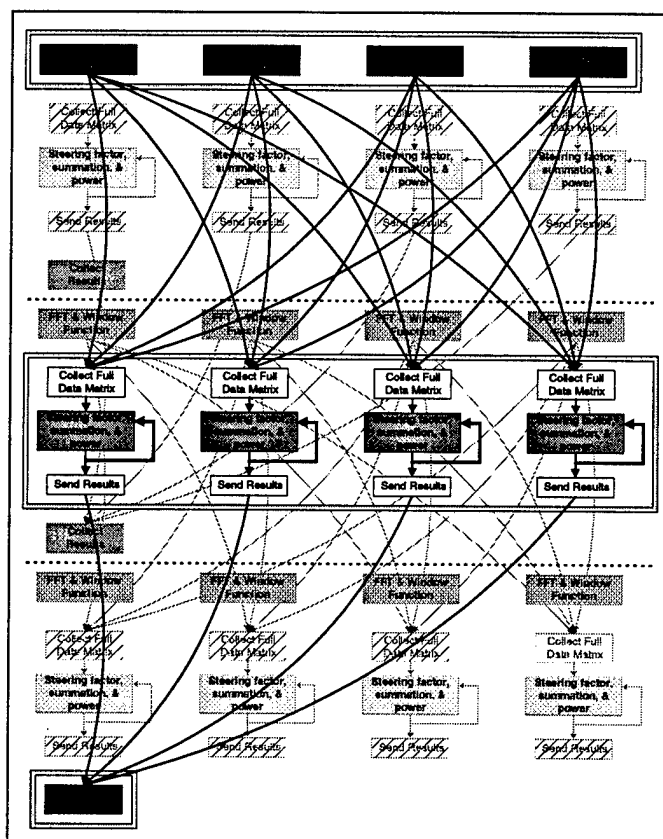


Figure B.15 - Pipelined Medium-grain Full-capability Network-independent FFT Beamformer Flowchart

The new medium-grain full-capability program was run over the UltraSPARC-I 8-node 155-Mbps ATM testbed with combinations of either 91 or 181 steering directions and either 4, 6, or 8 nodes. The non-pipelined medium-grain full-capability program, the coarse-grain full-capability

program, and the sequential program were also run. Figure B.16 shows the execution times for these programs when the beamformer is steering every 2 degrees and Figure B.17 shows the same results for steering every 1 degree.

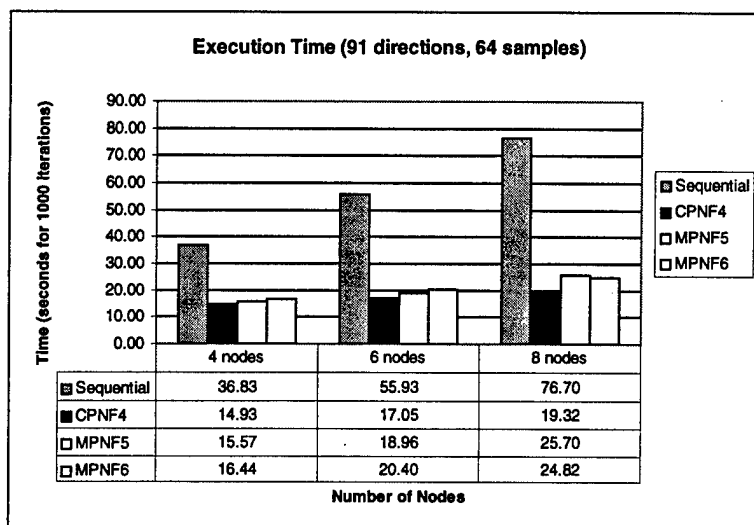


Figure B.16 - Execution Times for Pipelined Medium-grain, Non-pipelined Medium-grain, and Coarse-grain Programs with 91 Steering Directions

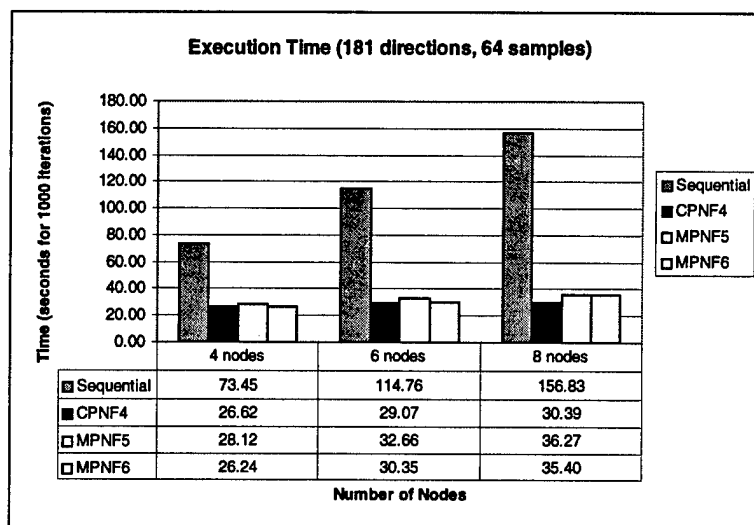


Figure B.17 - Execution Times for Pipelined Medium-grain, Non-pipelined Medium-grain, and Coarse-grain Programs with 181 Steering Directions

The speedups for these programs are shown in Figure B.18 and Figure B.19. As can be seen, the new pipelined program performs better than the non-pipelined version in most cases. The exception occurs for 91 steering directions and 6 or fewer nodes. In these cases, there is not enough data that must be communicated to warrant the extra complexity. In order to accommodate the pipelined communication, the MPI implementation must provide buffering for the data before it is received. Also, the beamformer is swapped out of the CPU so that the CPU can handle the incoming transmission. These delays do not overcome the propagation delay when small amounts of data are sent. In all other cases, the pipelined version performs better,

though not by much. This result is also explained by the fact that the MPI implementation must work harder when receives are not posted before the data actually arrives at the node. In fact, as the amount of data sent becomes greater, the implementation is put under further difficulty in managing the communication buffers. It will be seen later (when investigating the individual execution times of different categories) that the amount of time saved by not waiting in the receive call for propagation is partially negated by time the send call is flow-controlled and time the send call spends managing the link. The coarse-grained program still outperforms the medium-grained programs.

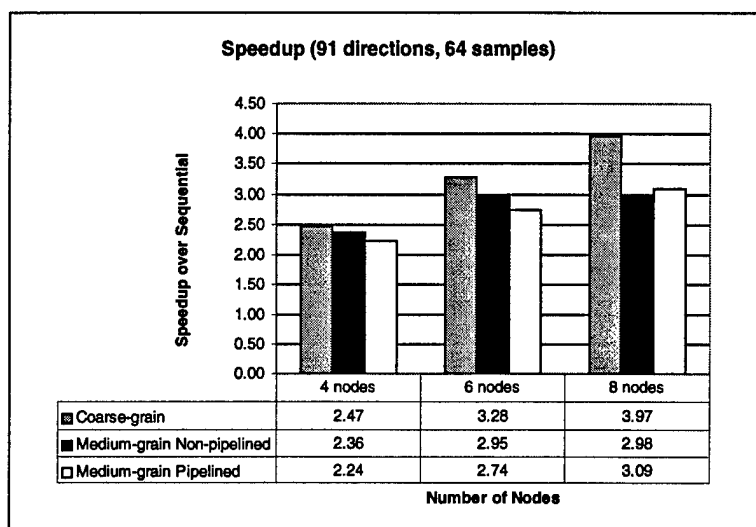


Figure B.18 - Speedups for Pipelined Medium-grain, Non-pipelined Medium-grain, and Coarse-grain Programs with 91 Steering Directions

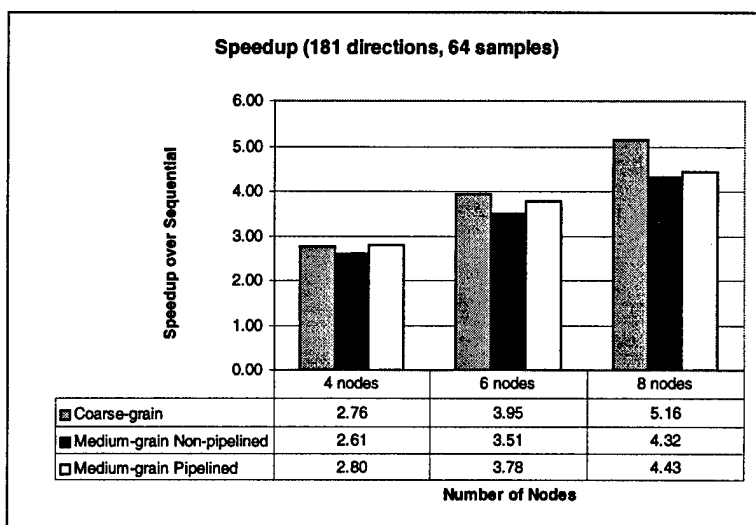


Figure B.19 - Speedups for Pipelined Medium-grain, Non-pipelined Medium-grain, and Coarse-grain Programs with 181 Steering Directions

These results show that the new version of the medium-grained program can provide an improvement over the non-pipelined version. The limitation is that the MPI implementation must buffer considerably more data than any of the other programs. In order for pipelined version to

work on the sonar array, a significant amount of memory must be provided. For arrays of 32 nodes or more, this may become impossible. Another requirement that is needed for the pipelined program to become useful would be a more efficient MPI implementation. Specifically, the sending node must be allowed to proceed almost immediately after a send call despite what hang-ups are occurring at the receiving end.

B.2.2. Category Definitions

Eight categories have been defined in the conventional FFT beamformer. Four of these categories are computational categories for different parts of the beamforming algorithm. Three categories are for communication. The last category is labeled "Other" and encompasses all the computation required to implement the program above that required in theory for the beamformer.

In order to clarify the algorithm computational categories, Figure B.20 shows the original sequential algorithm flowchart. The first computational category is called "FFT & Window" and includes the transforms for each node's data column and the multiplications of the data columns by the node-dependent shading factors. The "Steering Multiplication" category includes the multiplication of the data matrix by the complex delay factors. This category is represented by the single "Steering Factor Multiplication" block in Figure B.20. The next category is "Summation & IFFT," which includes the "Data Column Summation" block and part of the "Find Signal Power" block in Figure B.20. The last computational category for the algorithm is "Power Calculation," which encompasses finding the squared absolute value of the data samples and summing them (to include all frequencies in the directivity plot).

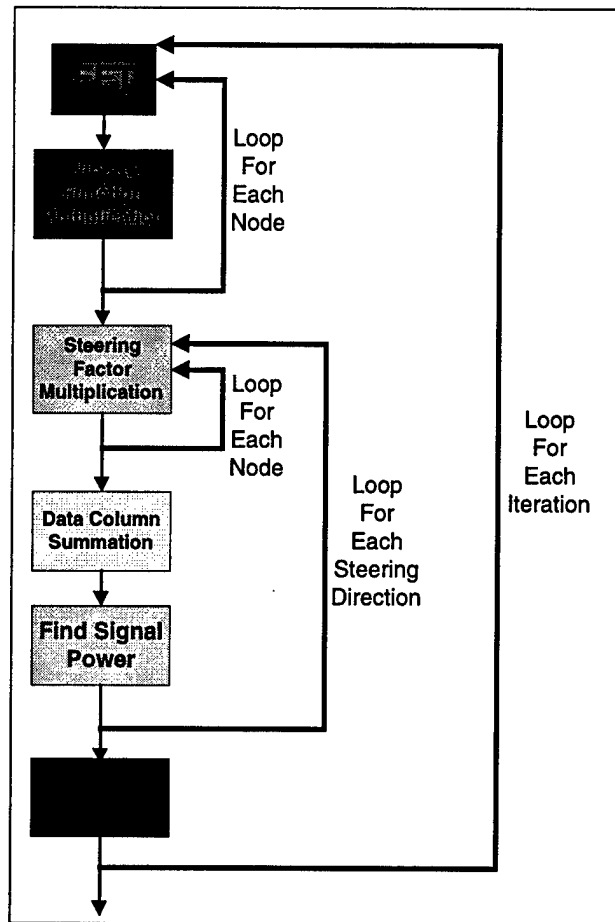


Figure B.20 – Sequential Flowchart for the Frequency-domain Delay-and-Sum Beamformer

The three communication categories are “Sending,” “Receiving,” and “Barrier.” The “Sending” category includes the time the parallel program spends in the *MPI_Send* call. This time includes the processing time required for the MPI implementation to manage its buffers or other such communication overheads. This time may also include the time waiting for a link to become free or the time waiting for the receiver’s buffer to have sufficient space. The “Sending” time does not refer to the time the data packet spends on the network. The “Receiving” category includes the time the program spends in the *MPI_Recv* call. This time includes the time the process spends waiting (blocking) for the desired data packet to arrive in addition to the implementation’s overhead. Again, this category is not meant to describe the time the received data packet had spent in the communication network. The “Barrier” category includes the time the process spends in the *MPI_Barrier* call. Since each process in the parallel system must start each iteration at approximately the same time (to ensure that a process does not get a jump start on the iteration before the iteration data is supposed to be available), each process issues a barrier after every iteration. These categories are omitted from the analysis of the sequential program.

The last category is “Other.” This category is measured by subtracting the other category times from the total time of program execution. The “Other” category includes overhead in the program that is not part of an MPI communication call or of the original theoretical beamforming algorithm. An example of processing time that gets included in this category is the CPU time used to calculate the steering direction in radians based on the starting direction in degrees and the steering direction loop index. Another example is the time required to calculate the destination and size for a data packet to be sent.

B.2.3. Category Performance Results

Each of the programs (including the sequential) has been augmented with short amounts of code to measure and record the execution time of the categories. In the case of the parallel programs, each process in the parallel system has its own independent measurements of the time it has spent in the different categories. For presentation purposes, these independent category timings for the different processes are averaged into a single set of per-processor category times for the parallel system.

The execution times for 1000 iterations of the sequential program, the coarse-grain full-capability program, and the two versions of the medium-grain full-capability programs are shown in Figure B.21 and Figure B.22.

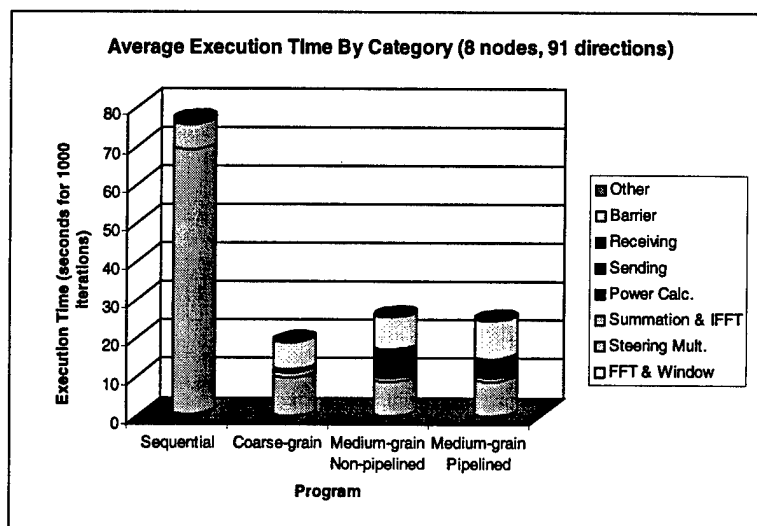


Figure B.21 – Execution Times for Sequential and Parallel Programs with 8 Nodes and 91 Steering Directions

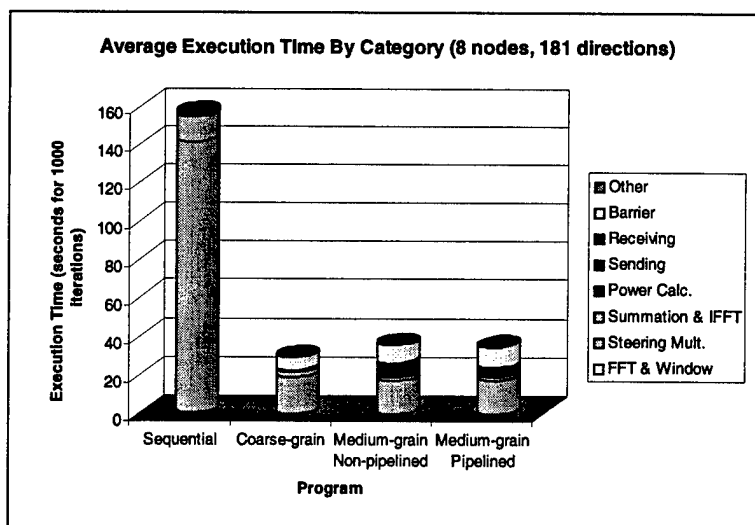


Figure B.22 – Execution Times for Sequential and Parallel Programs with 8 Nodes and 181 Steering Directions

There are a number of important details to note about the category results. First, the multiplication of the data matrix by the steering factors is the most considerable part of the total calculation. This category includes both the multiplication of the data by this factor, but also the calculation of the factor. This calculation involves several floating-point operations in itself. In fact, the performance of this category on the testbed was considerably worse than the expected performance based on the floating-point-operations (FLOPS) prediction models. In the original programs (before analysis via categories was done), this factor was completely recalculated for each different sample from each different node. In order to improve the performance of the category, only the part of the factor that changed from sample to sample and from node to node is recalculated. In this way, the programs calculate the portion of the factor that depends on node spacing, sampling frequency, and other unchanging variables and stores this portion in a variable. Then, when the programs loop for each node and for each sample, the variable is retrieved, the factor calculation is finished with via only two more floating-point operations, and the factor is multiplied by the sample. The performance savings this optimization provides are included in the above figures.

Second, the trade-off between the two communication methods in the non-pipelined version and the pipelined version are easily observed. In the non-pipelined version, after the communication has started, the nodes wait to receive the full data matrix. This wait appears in the "Receiving" category and is significant in the columns of the figures. In the pipelined version, the nodes do not spend as much time in the receive call because the data has already been received by the node and buffered. The receive does not need to block until the communication has propagated; it merely needs to retrieve the data from the buffer. Thus, the "Receive" category takes less time than it did in the non-pipelined program. On the other hand, the increased difficulty of managing the overlapped computation and communication and the increase in total amount of pending data in the communication system appears in the form of a longer execution of the send call. The "Send" category in the non-pipelined program takes less time to execute than the "Send" category in the pipelined version because the sender is flow-controlled when the receiver's MPI implementation is overwhelmed. In the end, the execution times between the non-pipelined and pipelined programs even out (though slightly in favor of the pipelined program).

Appendix C. Extensions to Advanced Beamforming Algorithms

C.1. Details of Algorithm Stages

The four stages of the split-aperture beamformer are Fourier Transform, beamform, cross-correlation, and mapping. Details of these stages are presented here.

C.1.1. Fast Fourier Transform Stage

If cross-correlation is implemented in the frequency domain without performing a sufficiently long FFT, an undesirable wrap-around effect will occur. This effect is a circular action that contaminates the original output of the correlation. To avoid the wrap-around effect in the cross-correlation stage, the input data needs to be zero-padded before the signal is passed to the FFT. The number of zeros depends on the length of the input data. The discrete cross-correlation function is defined in Equation C.1 where N is the number of samples in x and y .

$$c_{xy}(n) = \frac{1}{N} \sum_{i=0}^{N-1} x(i)y(i+n) \quad n = 0, 1, \dots, N-1$$

Equation C.1

In the frequency domain, this equation translates to {Equation C.2}.

$$C_{xy}(k) = X(k) \times Y(k)^* \quad k = 0, 1, \dots, 2N-2$$

Equation C.2

According to Equation C.1, the length of the cross-correlation result will be $N+N-1$, which is the same length as the convolution of the two inputs; therefore, to prevent the wrap-around effect, x and y should be zero-padded to have FFT lengths greater than $N+N-1$.

With lengthy FFT sets, the computational cost of the algorithm severely increases. There are two methods that can be done in the FFT stage to reduce this side effect. One is frequency-bin averaging, and the other is ignoring the wrap-around effect. The frequency-bin averaging technique involves taking the average value between several consecutive FFT samples, thus decreasing the total number of FFT samples. This procedure creates a FFT sample set with fewer samples, and additionally, can make the number of FFT samples less than the sample length of the original time-domain representation. This method is very useful for beamforming algorithms that use the Cross-Spectral Matrix (CSM), such as the adaptive beamforming algorithm explained later in this document. Ignoring the wrap-around effect is also possible because we are only interested in the small amount of time delay in the cross-correlation at each steering angle, and the wrap-around effect starts occurring for large time delays. In the final stage, the mapping process, we can discard the corrupted samples and the final output remains correct.

C.1.2. Frequency-Domain Beamforming Stage

Frequency-domain beamforming in SA-CBF is basically the same as single-aperture frequency-domain beamforming except the phase centers of each sub-array are considered. In

this stage, the plane wave replica vector, v_{jm} , multiplies with the input transformed data; hence, the input data of node m is steered to the specific direction j in relation to the phase center. The phase center is the reference point used to calculate the cross-correlation. The replica vector is defined as $v_{jm} = e^{i\omega\tau_{jm}}$, where τ_{jm} , Equation C.3, is the relative time delay between the phase center (Equation C.4) and node m . To get the linear time delay, τ_{jm} , we need to steer the angle nonlinearly in constant increments of $\sin(\theta)$. In the equations below, r_m is the node position, c_{sound} is the speed of sound in water, N_{Fk} is the first index number of sub-array k , N_{Lk} is the last index number of sub-array k , N_k is the number of the nodes in sub-array k and θ_{sub-j} is the steering angle of the subarray.

$$\tau_{jm} = \frac{\sin \theta_{sub-j} \times (r_m - c_k)}{c_{sound}} = \frac{d_{jm}}{c_{sound}}$$

Equation C.3

$$c_k = \frac{1}{N_k} \sum_{m=N_{Fk}}^{N_{Lk}} r_m$$

Equation C.4

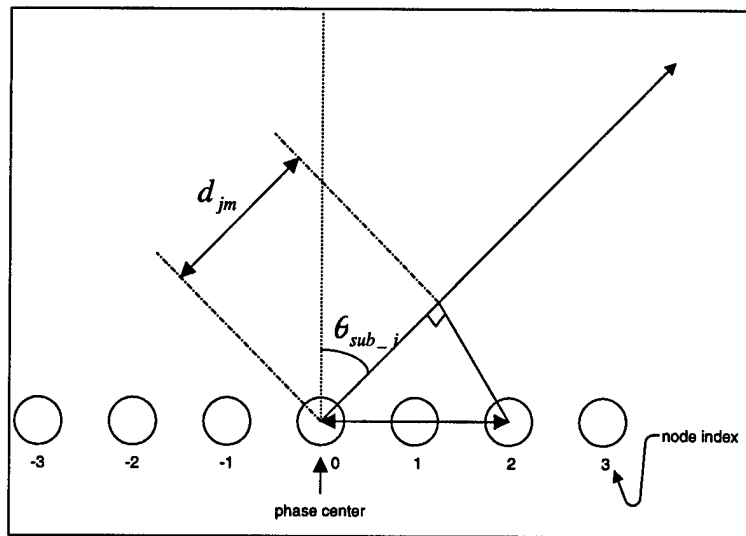


Figure C.1 - Geometry of the sub-array elements

If we properly steer the beamformer to an incoming signal, the multi-channel input signals will be amplified coherently, maximizing power in the beamforming output; otherwise, the output of the beamformer is attenuated to some degree.

Unlike the single-aperture beamforming algorithm, the Split-Aperture algorithm does not need to steer at every individual desired angle. The cross-correlation causes some redundant information between the adjacent steering angles so each angle generates a range of the time

delay plot. The number of output bearings is two or four times the number of beams that are formed at each sub-array because we can use τ -interpolation to increase the resolution of the output. The τ -interpolation procedure will be explained in the mapping process stage.

C.1.3. Cross-correlation Stage

Once we complete the frequency-domain beamforming for each steering angle, cross-correlation is used to detect the time delay between the two phase centers. The maximum peak of the cross-correlation indicates the time delay of the two beamformer outputs. Additionally, the time axis of the cross-correlation can be transformed to an angle axis by the simple geometric transformation in Equation C.5. Figure C.2 helps explain the derivation of this equation. Note that τ_c is the time-axis value of the cross-correlation, d_{12} is the distance between phase centers, and θ_j is the corresponding angle of the final beamforming plot.

$$\theta_j = \arcsin\left(\frac{d_j}{d_{12}}\right) = \arcsin\left(\frac{\tau_c \times c_{\text{sound}}}{d_{12}}\right)$$

Equation C.5

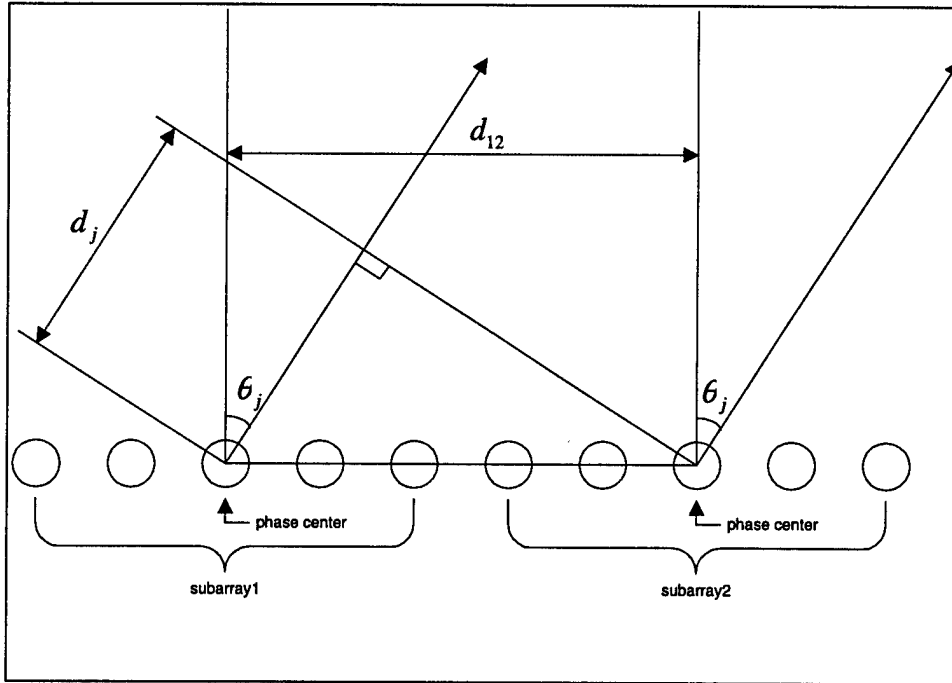


Figure C.2 - Geometry of the whole array with 10 nodes.

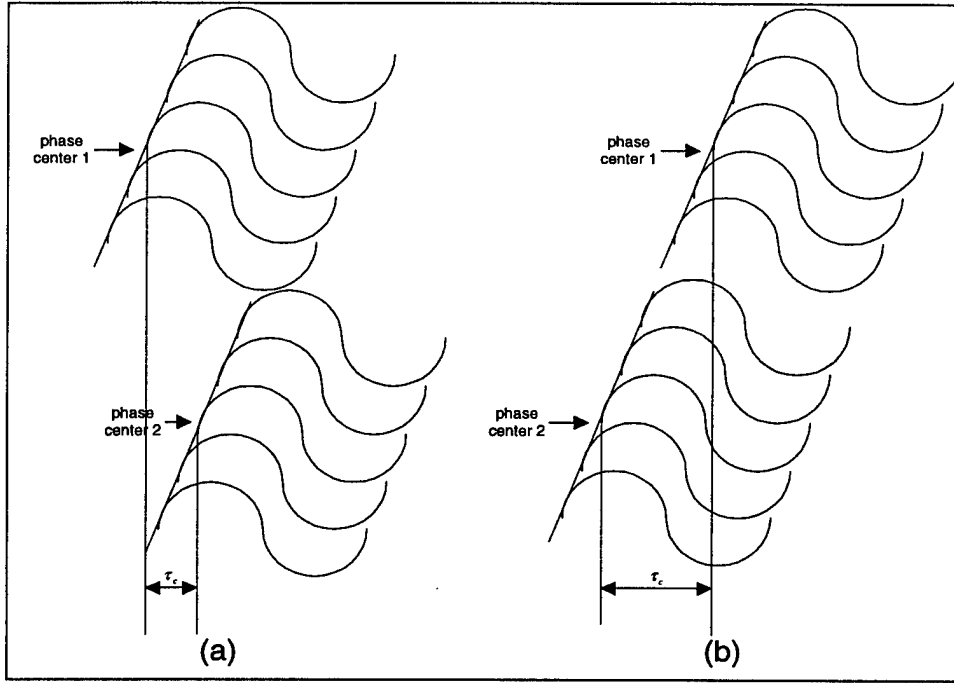


Figure C.3 – Illustration of the Cross-correlation Correctness. The cross-correlation depends on the time delay τ_c ; (a) is incorrect because $\theta_{sub_j} \neq \theta_j$ and (b) is correct, $\theta_{sub_j} \approx \theta_j$.

We are only interested in the small number of angles or time delays in the cross-correlation adjacent to the beamforming angle of the sub-array (see Figure C.3(b)) since sub-array nodes are already steered to the specific direction. Beyond this range of angles (see Figure C.3(a)), the cross-correlation is not correct so that portion of the cross-correlation must be eliminated from the final output to maintain correctness.

Before the inverse FFT is applied to obtain the cross-correlation as function of time delay, the Smoothed COherent Transform (SCOT) is performed for more distinct shaping of the cross-correlation. Fundamentally, SCOT sets the magnitude portion of the cross-correlation in the frequency domain to 1. This procedure results in a better resolution in the output plot because a wide bandwidth in the frequency domain corresponds to a shaper image in the time domain. This spectral whitening is accomplished either by taking the instantaneous magnitude of the cross-correlation in frequency or a running average of the magnitude, which is calculated with the previous magnitude of the data. Sometimes, a spectral window is used to acquire the smooth beamforming plot. SCOT weighting is calculated according to Equation C.6 and Equation C.7.

$$\bar{A}_m(q) = (1 - \alpha)\bar{A}_m(q-1) + \alpha|C_{xy}|(q)$$

Equation C.6

The average at time q is based on the average at the previous time $q-1$. If $\alpha = 1$, no running average is done. Finally, SCOT weighting divides the cross-correlation by this factor.

$$\overline{C}_{xy} = \frac{C_{xy}}{A_m}$$

Equation C.7

Finally, a normalizing factor, which is the integral of the magnitude coherence, is used. This factor divides the SCOT result and ensures a normalized function with magnitude less than unity.

C.1.4. Mapping Process Stage

There are two methods of mapping the two cross-correlations to the final output correlation. The first method is to apply the weight function to the individual beam correlations with some overlap of neighbor correlations, and then add up all the cross-correlation values time-by-time. The weight function center is placed at the steering angle of the sub-array so that we take only the accurate values from each correlation (see Figures C.4 and C.5).

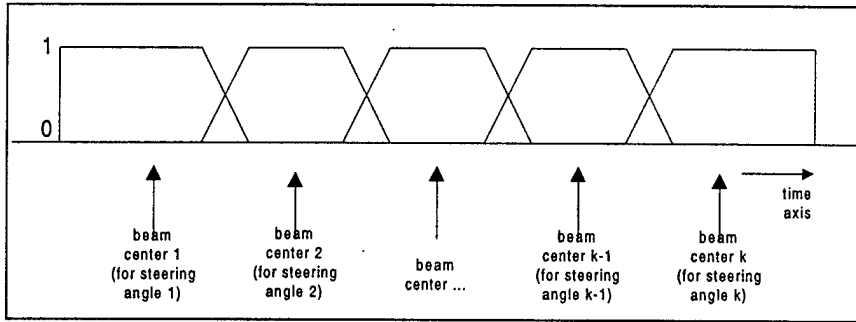


Figure C.4 – Weight Functions for Composite Beam Correlation

The other method is τ -interpolation, which involves taking only a range of cross-correlation values, $c(\tau_c)$ where $\tau_c = f(\theta_{sub_j} \approx \theta_j)$ (see Figure C.4 and Equation C.5) and then working with raised-cosine weights to figure out the interpolated angle from the two adjacent steered angles. Figure C.5 shows steered angles as long rays and interpolated angles as shorter rays.

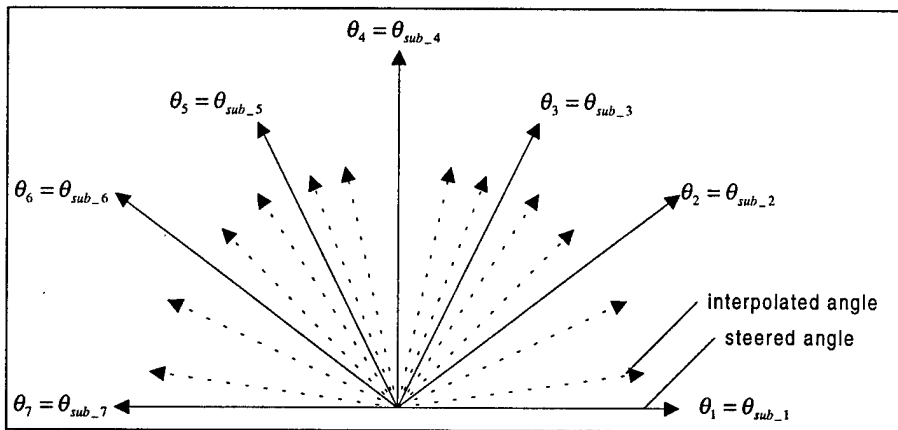


Figure C.5 - Representation of Beamformer Angles, Steered Angles, and Interpolated Angles

In this case, only some range of cross-correlation values are required to calculate the final beamforming plot. The inverse discrete Fourier transform is then performed using Equation C.8 rather than by using a conventional inverse FFT. This method will decrease the computational cost of the inverse FFT. Note that in Equation C.8, M_1 and M_2 are the minimum and maximum frequency bin numbers in which we are interested, \tilde{C}_m is the SCOTed and normalized frequency-domain cross-correlation, and Δf is the frequency resolution of the FFT.

$$c(\tau_j) = \frac{1}{N_{FFT}} \sum_{m=M_1}^{M_2} 2\text{Re}[\tilde{C}_m e^{i2\pi(m-1)\Delta f\tau_j}]$$

Equation C.8

The inverse discrete Fourier transform (Equation C.8) is somewhat different in form than the conventional inverse FFT because we want to detect the time delay between phase centers. The distance between the phase centers is large compared to the distance between nodes. The cross-correlation function can detect only one cycle of the time delay for the specific frequency. As the frequency increases, the wavelength will be shorter and the detectable time delay will be decreased. Even if we detect some time delay between the phase centers, this value might not be correct because several cycles of wave have already passed in the distance. To avoid this problem, we apply envelope correlation in the inverse FFT stage. The basic concept is that a band-passed signal is demodulated to base-band before taking the IFFT. As can be seen in Equation C.8, the real value of \tilde{C}_m over some range ($M_1 \sim M_2$) is used because of the demodulation and band-passing. The factor of 2 is because we process only the positive frequency extent.

For the output angle θ_{out_j} , two correlation functions will be evaluated in the closest beam pair $c_L(\tau_j)$ and $c_R(\tau_j)$. The τ -interpolation is defined in Equation C.9 with its component variables defined in Equation C.10. In these equations, θ_L and θ_R are the beamformed angles to the left and right, respectively, of the output angle θ_{out_j} . Using this τ -interpolation method, the final display of the beamformer has less variance in the value as shown in the following equations.

$$c(\theta_{out_j}) = h_L c_L(\tau_j) + h_R c_R(\tau_j)$$

Equation C.9

$$h_L = \frac{1}{2} \left[1 + \cos \pi \left(\frac{\theta_L - \theta_{out_j}}{\theta_L - \theta_R} \right) \right] \quad ; \quad h_R = 1 - h_L$$

Equation C.10

C.2. Computer Simulation

C.2.1. Parameters Used in the Simulation

- $M = 18$; Number of nodes
- $F = 30$; Processing frequency band $0.2F$ - $3.0F$
- $C = 1500$; Speed of sound (m/s)
- $L = C/F$; Wavelength (m)
- $D = 0.45*L$; Spacing between nodes (m)
- $FS = 12*F$; Sampling frequency (Hz)
- $Numsamp = 128$; Number of samples
- $Sub = 2$; Number of sub-arrays
- $Fnum = 256$; Number of FFT points for each nodes
- $Nangle = 15$; Number of angles steered

C.2.2. Generate Input Data

One of the biggest problems in the simulation environment is how to generate input data. Time sequences are necessary for the beamforming simulation. One has to be concerned not only with the time delay between node data but also with the spectrum characteristics. To show the broadband property of the beamformer, band-limited plane wave impulses are used as input data. The band-limited impulse can easily be created in the frequency domain. Figure C.6 shows this method. First, in the frequency domain, a signal is created that has a constant non-zero value around the processing band and is zero elsewhere. Second, the inverse FFT is applied to the frequency data to obtain the time series data. Last, the process is repeated with the node-specific shift factor for the every node until all nodes' data is created. The time shift in the time domain can be described as a multiplication with some complex exponential value as shown in Equation C.11. The time delay can be calculated using Equation C.3 above with any phase center, as long as the phase center is kept constant throughout the data generation process. The result of the signal generation for the various nodes is shown in Figure C.7.

$$x[n - n_d] \Leftrightarrow e^{j\omega n_d} X(e^{j\omega})$$

Equation C.11

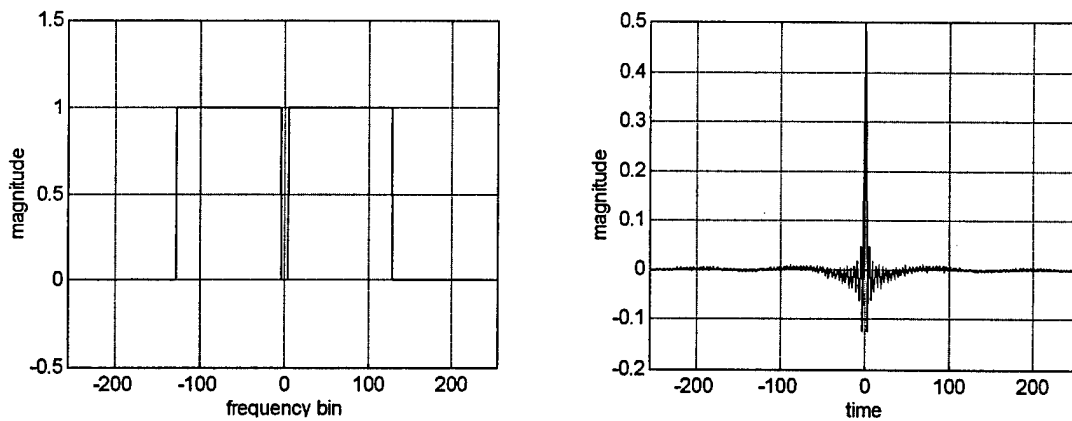


Figure C.6 - Demonstration of the frequency and time domain relation.

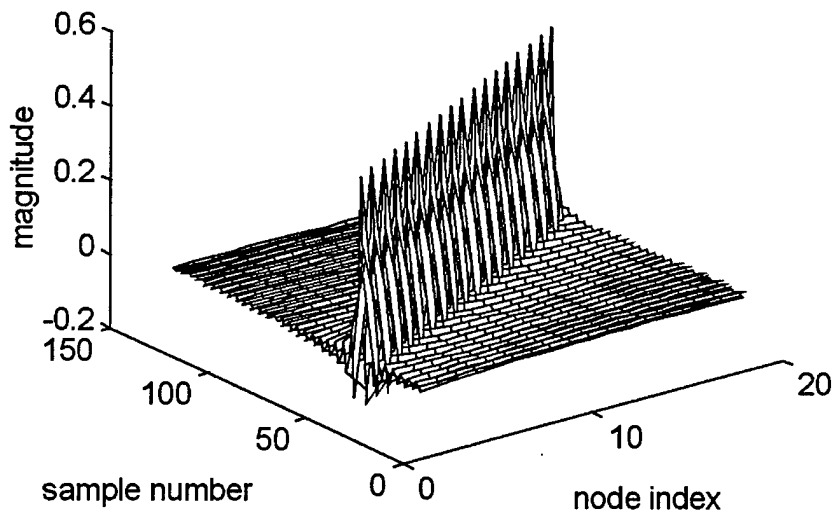


Figure C.7 – Generated Input Data without Noise from 47.8 degrees.

C.2.3. Complex Situations

To this point, we have tested the SA-CBF in the ideal situation: without noise and with one single source. However, once the sonar array is launched, its environment will be nothing like the ideal one. To see some other aspects of the algorithm, harsher conditions must be used in experimentation. One of the important performance factors in the beamformer is the resolution, that is, the ability to distinguish different sources that have distinct incoming angles.

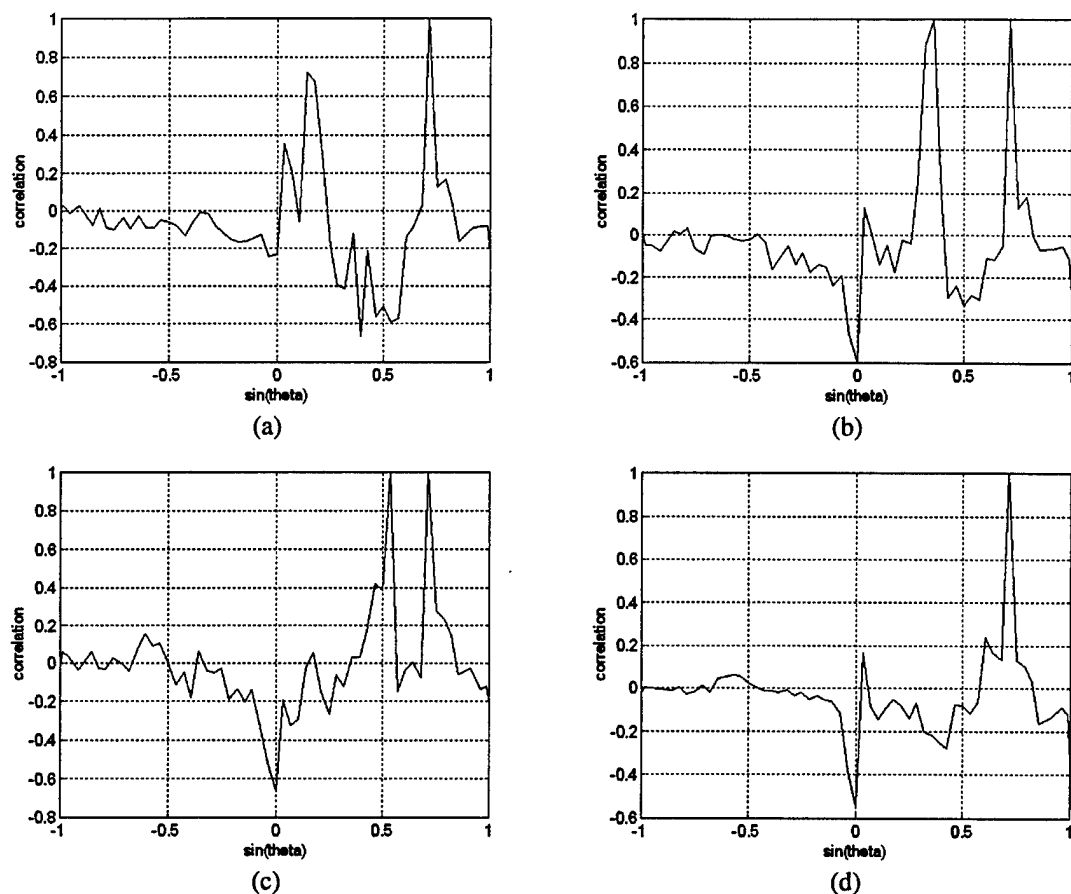
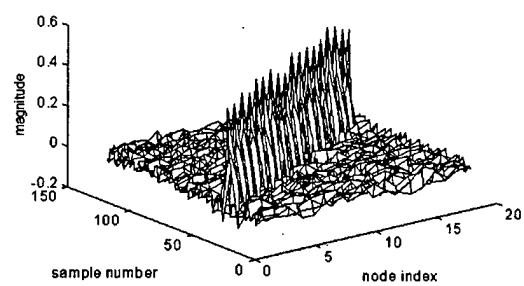


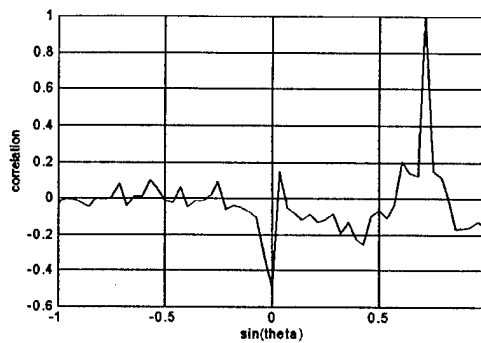
Figure C.8 - SA-CBF Output for Multiple Sources. Each plot has 15 steering angles and 57 output angles. The signal sources are located at: (a) 10.7°, 47.8°; (b) 21.7°, 47.8°; (c) 33.8°, 47.8°; (d) 43°, 47.8°.

In the above figures, looking at the beamformer output, one can distinguish the directions of arrival if the angles are quite far apart, for example Figures C.8(a)-(c); otherwise, the final display looks like one source (Figure C.8(d)). This result occurs because the split-aperture algorithm cannot interpolate targets between sub-array steering directions when those sub-arrays cannot themselves distinguish the targets. The resolution can be improved by increasing the number of steering angles in the sub-arrays; however, this solution causes an increase in computational cost.

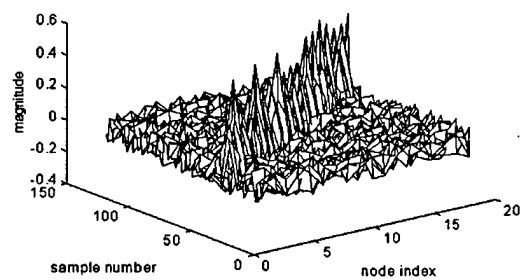
The other side of the performance issue that must be addressed is the noise response of the beamformer. Due to the sensor and the environment, there will be some noise in the signal. Lessening the noise effect is accomplished by several techniques including SCOT, running time averaging, etc. The following results show the beamformer output in noisy conditions.



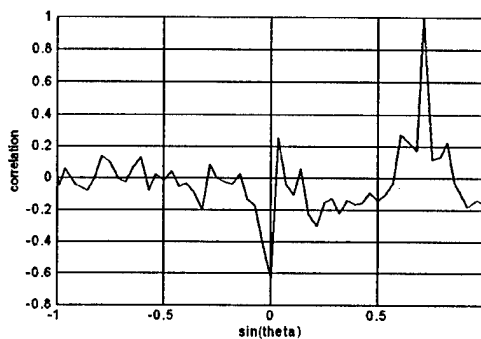
(a)



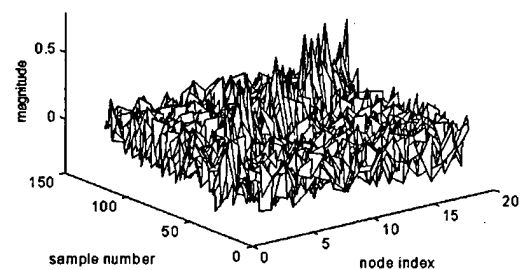
(b)



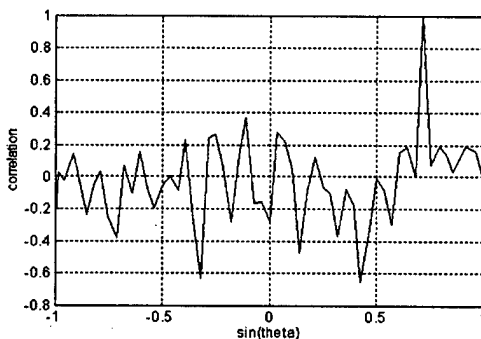
(c)



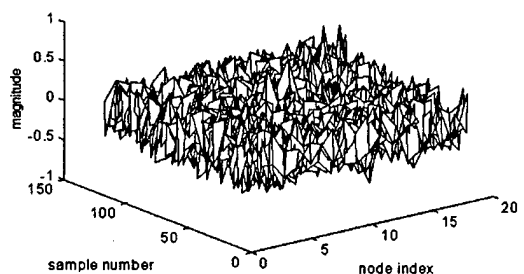
(d)



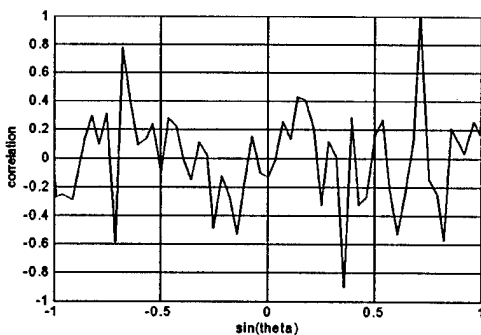
(e)



(f)



(g)



(h)

Figure C.9 – Input Signals with Noise and SA-CBF Outputs. In these graphs, the desired signals all have max value ≈ 0.5 . (a) signal with white noise ($-0.05 \sim 0.05$), (b) output of the beamformer, (c) signal with white noise ($-0.1 \sim 0.1$), (d) output, (e) signal with white noise ($-0.2 \sim 0.2$), (f) output, (g) signal with white noise ($-0.4 \sim 0.4$), (h) output.

As more noise is added to the signal, the side lobe heights in the output of the beamformer are increased. If we do not apply SCOT to the beamformer, the shape of the beamformer results will get worse. Note that in Figure C.9, the running-average scheme is not used to create the beamforming results and that if a running average been used, the beamformer contrast would have been increased, even with severe noise.

Appendix D. Extensions to Prototype Hardware Architecture

D.1. Additional Node Processor Features

The C54 provides seven different addressing modes. They are immediate addressing, absolute addressing, accumulator addressing, direct addressing, indirect addressing, memory-mapped register addressing, and stack addressing. Program memory addressing is controlled by the PC, which is used to fetch instructions sequentially. Conditional branches and repeat instructions change the PC by adding to or subtracting from the current value through an ALU. When functions or interrupts occur, the current PC is saved onto the stack using the stack pointer (SP), and the appropriate routine is serviced. The PC value is restored from the stack and normal instruction execution resumes.

Several reasons have been presented for using DSP hardware, as opposed to general-purpose hardware, for increased system performance. However, there are similar characteristics between the two, such as their use of pipelines that increase performance in both architectures. Pipelines increase the rate of executing instructions by using parallel hardware to implement different stages of an instruction. Ideally, pipelines speed up instruction execution by a factor nearly equal to the number of pipeline stages in the system. This feature may optimize the implementation of parallel beamforming algorithms. The six-stage instruction pipeline used in C54s allows at most six instructions to be active in any given cycle of execution. The six stages of the pipeline are: *prefetch, program fetch, decode, access, read, and execute*.

The analog interface circuit on the DSKplus provides one channel of voice quality data acquisition. This acquisition is made possible through A/D and D/A signal conversions with 14 bits of dynamic range for linear resolution; a programmable anti-aliasing filter for optimized filter implementations; software programmable sampling rates; reset, feedback, loopback, and low-power modes of execution; an auxiliary input with software-selectability; two-channel analog input summing capability; and optional master/slave configurability for cascading. The analog interface circuit connects to the C542 TDM serial port directly. To send and retrieve data from the TDM port, the analog circuit generates pulses specified by shift clock (SCLK) and frame sync (FS). A 10-MHz master clock determines the timing specifications for the pulses generated by the SCLK and FS. The DSKplus host port interface (HPI) is a programmable array logic device, PAL22V10 (PAL[®]), that interfaces the host PC's parallel port and the C542 HPI port. This device allows the host computer to configure the DSKplus to operate in several different parallel transfer modes, such as 4-bit unidirectional and 8-bit bidirectional schemes. The connection allows the DSKplus to communicate through the parallel port of the PC [TMS96].

The TDM serial port allows the execution of time-division multiplexing. Time-division multiplexing is the division of time intervals into subintervals that behave as communication channels. Multiple devices can be connected to a target device using the communication channels, thus allowing communication links between devices. The TDM serial port is accessed through the JP1 header on the DSKplus board. The TDM feature of the C542 processor supports the network topology that will be used in the prototype architecture.

D.2. TDM Serial Port

As previously mentioned, the eight DSKplus boards will be connected through the JP1 header, which provides the interface to the TDM serial port. The time-division multiplexed

(TDM) serial port allows a C54x board to communicate with up to 7 other boards, therefore there can be up to eight communication channels. Each channel allows for one 16-bit data transmission from a single device but any number of devices can receive the data.

D.2.1. TDM Serial Port Registers

The TDM port is user controlled through six memory-mapped registers and two other registers (TRSR and TXSR) that implement the double-buffering capability. The eight registers are listed in Table D.1. The layout of the six registers accessible by the user is shown in Figure D.1.

Register	Description
TRCV	TDM data receive register
TDXR	TDM data transmit register
TSPC	TDM serial port control register
TCSR	TDM channel select register
TRTA	TDM receive/transmit address register
TRAD	TDM receive address register
TRSR	TDM data receive shift register
TXSR	TDM data transmit shift register

Table D.1 - TDM Serial Port Registers. The TDM Serial Port is controlled through eight memory-mapped registers.[TMS97A]

- The TDM data receive register (TRCV) holds the data sampled from the TDAT wire (see Figure D.2).
- The TDM data transmit register (TDXR) holds the data the transmitting device is to transmit serially through the TDAT wire.
- The TDM serial port control register (TSPC) contains various bits that control the operation of the TDM port.
- The TDM channel select register (TCSR) is used to specify the time slot each device is to transmit.
- The TDM receive/transmit address register (TRTA) holds the receive address of each device and the addresses it is to transmit to during its time slot.
- The TDM receive address register (TRAD) holds various bits regarding the status of the TADD line (see Figure D.2).
- The TDM data receive shift register (TRSR) controls the storing of the received data from the TDAT line to the TRCV.
- The TDM data transmit shift register (TXSR) controls the transfer of the transmit data from the TDXR to the TDAT line.

Reg.	Addr.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRCV	0030h	Receive Data															
TDXR	0031h	Transmit Data															
TSPC	0032h	Free	Soft	X	X	XRDY	RRDY	IN1	IN0	RRST	XRST	TSM	MCM	X	0	0	TDM
TCSR	0033h	X	X	X	X	X	X	X	X	CH7	CH6	CH5	CH4	CH3	CH2	CH1	CH0
TRTA	0034h	TA7	TA6	TA5	TA4	TA3	TA2	TA1	TA0	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
TRAD	0035h	X	X	X2	X1	X0	S2	S1	S0	A7	A6	A5	A4	A3	A2	A1	A0

Figure D.1 - TDM Serial Port Register Bits. [TMS97A]

D.2.2. TDM Serial Port Operation

The TDM serial port is implemented using a four-wire bus consisting of clock (TCLK), frame (TFRM), data (TDAT), and a wire (named TADD) that carries device address information. The TDAT line is formed by an external connection of the TDX and TDR signals from the '54x device. The TCLK line is formed by an external connection of the TCLKX and TCLKR signals from the '54x (see Figure D.2).

The TADD line is a bidirectional signal that determines which device(s) should execute a receive operation for a particular time slot. The data is transmitted on the bidirectional TDAT line. The TFRM and TCLK signals synchronize all TDM port operations and are generated by only one of the devices in the system.

Only one device controls the TADD and TDAT lines per communications channel or time slot. The rest of the devices sample these lines to determine if there is valid data to be read by any of the devices on the bus. The address of the intended receiving device is put on the TADD line and the data to be transmitted is put on the TDAT line by the transmitting device. When a device recognizes its address (specified by the lower eight bits of TRTA, RA7 – RA0), it reads the TDAT line and the data is transferred from TRSR to TRCV. When a valid receive occurs, a receive interrupt (TRINT) is generated indicating that TRCV can be read.

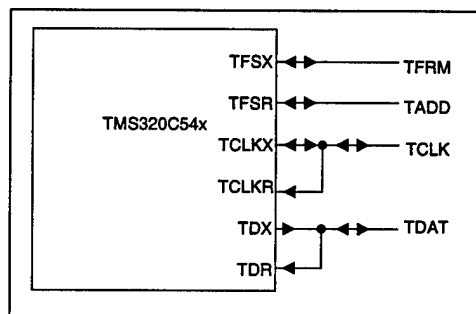


Figure D.2 - TDM Serial Port Wiring Diagram. The TDX and TDR pins are externally connected to create the TDAT line. [TMS97A]

The source device for the TCLK and TFRM signals is set by the MCM and TXM bits, respectively, of the TSPC registers. Only one device can have these bits set to 1 at any given time, usually specified during initialization, and this device provides the clock and frame signals. The TCLK frequency is one-fourth the CLKOUT frequency, and the TFRM pulse occurs every 128 TCLK cycles. These signals could be driven by external sources if the MCM and TXM bits are set to 0 on all the devices.

The status of the TADD line can be determined by reading the contents of the TDM receive address register (TRAD). Bits 11-13 (X0-X2) indicate the current time slot number. Bits 8-10

(S0-S2) indicate the last slot number to receive data plus one. Bits 7-0 (A7-A0) hold the last address placed on the TADD line.

D.2.3. Receive/Transmit Operations

A given device can specify which time slot(s) it is going to transmit in by setting one or more of the eight bits of the channel select register (TCSR) to 1. Only one device can transmit per time slot but a device can transmit during multiple time slots.

The lower half of the receive/transmit address register (TRTA) specifies the receive address of the device and the upper half specifies the transmit address. The receive address is the value it compares (by a logical, bit-wise AND) to the transmit address on the TADD line during a transmission to determine if it should receive the data on the TDAT line. The receive address consists of a 1 in any of bits RA0-RA7 (see Figure D.1) of TRTA. The transmit address, TA7-TA0, is the value the transmitting device places on the TADD line during a slot that specifies which devices are to execute a receive.

In the following example, which was adapted from [TMS97A], of a TDM configuration, eight devices communicate with one another using their respective channel numbers to transmit. The specific communication pattern is described by Figure D.4. TRTA specifies the receive and transmit addresses, and TCSR specifies during which channel each device will transmit.

Note that in Figure D.3 each device has a unique receive address (specified by RA7 – RA0); this allows the transmitting device to transmit to one or all the other devices by simply changing the transmit address (TA7 – TA0). Each device's channel of transmission is specified in bits 7-0 of TCSR.

TRTA

			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Dev.	TA	RA	TA7	TA6	TA5	TA4	TA3	TA2	TA1	TA0	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
0	80h	01h	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	40h	02h	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
2	20h	04h	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
3	10h	08h	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
4	08h	10h	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
5	04h	20h	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
6	02h	40h	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
7	EFh	80h	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0

TCSR

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Dev.	X	X	X	X	X	X	X	X	CH7	CH6	CH5	CH4	CH3	CH2	CH1	CH0
0	X	X	X	X	X	X	X	X	0	0	0	0	0	0	0	1
1	X	X	X	X	X	X	X	X	0	0	0	0	0	0	1	0
2	X	X	X	X	X	X	X	X	0	0	0	0	0	1	0	0
3	X	X	X	X	X	X	X	X	0	0	0	0	1	0	0	0
4	X	X	X	X	X	X	X	X	0	0	0	1	0	0	0	0
5	X	X	X	X	X	X	X	X	0	0	1	0	0	0	0	0
6	X	X	X	X	X	X	X	X	0	1	0	0	0	0	0	0
7	X	X	X	X	X	X	X	X	1	0	0	0	0	0	0	0

Figure D.3 – TDM Register Contents: (a) TRTA is the TDM Receive/Transmit Address Register and (b) TCSR is the TDM Channel Select Register.

This configuration corresponds to the following scenario:

Channel	TADD	Transmitter Device	Receiver Device(s)
0	80h	0	7
1	40h	1	6
2	20h	2	5
3	10h	3	4
4	08h	4	3
5	04h	5	2
6	02h	6	1
7	EFh	7	0-6

Figure D.4 - TDM Communication Scenario

In this example, it can be seen how any number of devices can receive data during any given time slot. The broadcasting capability is an important feature in the design of the prototype.

The TDM port offers the flexibility of communication among processors needed to implement the prototyping of the sonar array. Its broadcasting and simple configuration capabilities help reduce communication overhead.

Appendix E. Fault-tolerant Architectures and Algorithms

The realization of a real-time sonar array beamformer imposes particular fault recovery techniques. These techniques will eliminate the need for a robust network transmission control protocol with an advanced ARQ (automatic repeat request) mechanism. In a real-time system, these methods are simply too slow and unneeded. The particular needs of this project relate well to a Global Data Scope (GDS) type system. GDS characterizes the scope of the sample set for a particular distributed parallel sonar array (DPSA).

Preliminary research suggests that a GDS system is quite realizable and perhaps practical. The alternative is a Local Data Scope (LDS) system in which a sample set is particular and local to each node. The GDS system simply updates a sample set on each node's memory by utilizing broadcast mechanisms. Therefore, each node has a continually updated local set of each node's sample set. This is quite feasible in an array of 32 nodes, utilizing approximately 2 Mbps of network resources if the network supports broadcast sends. The advantages of this type of system include a very deterministic traffic pattern, easy parallel decomposition, and numerous fault-tolerance benefits. However, there are always disadvantages. The GDS system certainly will use much more memory than the LDS approach. This problem becomes apparent when observing that data is replicated at each node. Secondly, a pre-processing stage such as an FFT is difficult to implement before replicating data to each node; however, it is feasible. But it still maintains its greatest value: efficiency through simplicity.

E.1. Fault-Tolerant Services

Fault tolerance is an obvious requirement for the Distributed Parallel Sonar Array. The first justification is (since the device is battery powered in distribution) the fact that the system is guaranteed to fail at some time at an arbitrary node or nodes. Therefore, the probability of failure is 100 percent. Since the system is also autonomous, it is impossible for human interaction to reset the array. However, with such an ambitious system, one or a few disabled nodes need not handicap the entire array. A list of likely causes of system failure might be network failure, node failure, poorly adapted code, broken communication links, etc. This project discusses and attempts to solve software fault-tolerant issues. These mechanisms may be thought of as a kernel providing a buffer between the network (the only means of communication) and the beamform application. This kernel could provide various services such as higher network protocol layers (i.e., a transport control), which may selectively provide automatic repeat requests, error control, etc. However, a few assumptions were made about the system that must be supported in the hardware. The most important of these requirements is that the nodes must have the ability to communicate when other nodes have failed. With the constraint of a linear array wired with point-to-point links, satisfying this assumption is hardly trivial. The basic assumption, then, is that any node may communicate although nodes in its path may be disabled. This requirement is satisfied by the optical bypass switch already created in phase one.

The Fault-Tolerant Kernel is a software protocol that may be placed in between the network protocol layers and the beamform algorithm. The kernel offers services to the application, which improves the fault tolerance of the system. In particular, five fault-tolerant measures were built into the kernel to support the software on top of the Global Data Scope (GDS) system. The first of these measures is the ability of nodes to pad data streams if data was corrupted or lost by the network. Second, each important transaction, such as a system request, is built with a handshaking protocol, which ensures delivery. Third, periodic sensing is employed, where "request ping" messages are sent from the master node to each of the stations. These messages also allow the master node to allocate jobs to the existing slave pool and reallocate when

necessary. Fourth, a mechanism was created to resynchronize the array and the circular buffer and sample pointers (which are discussed in detail below). Last, each slave node also periodically checks the status of the master node. If the master fails, each node has the capability of distributing jobs to each of the other slaves. In other words, any node may become master.

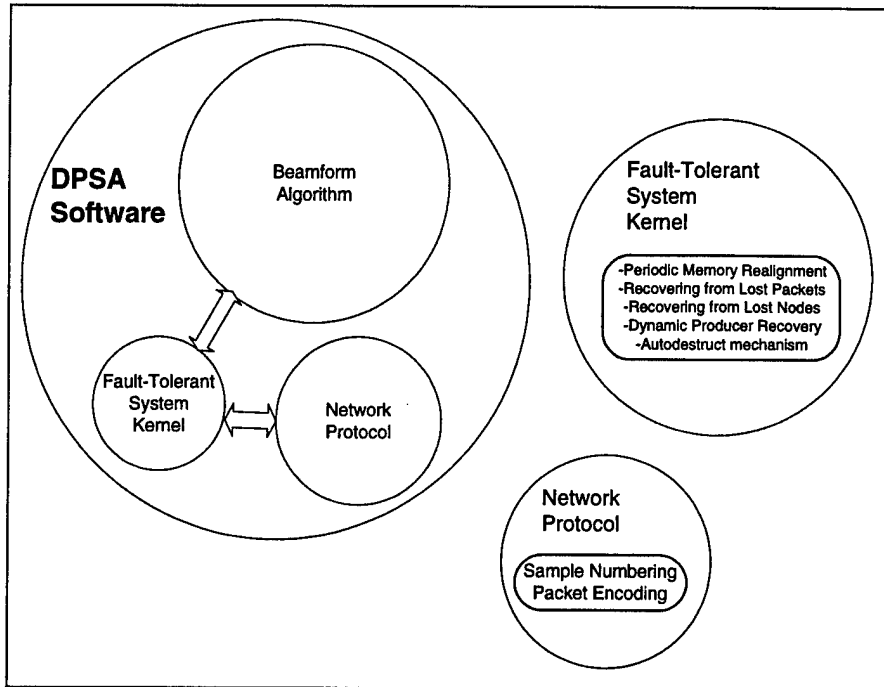


Figure E.1 - System Diagram and Fault-Tolerant Services. The Fault-Tolerant Kernel is a buffer between the Network and higher application process. It provides services such as higher network protocols, as well as mechanisms particular for a DPSA.

This fault-tolerant research was implemented on a simple time-domain beamform algorithm which is also called GDS. The GDS beamform algorithm (for clarity, this will be referred to as GDS-BA) defined a new type of parallel decomposition, which again may be called Global Data Scope (GDS). The idea of the GDS-BA system is to distribute data to each of the nodes so that the beamform algorithm, in whatever domain necessary, may be easily realizable with a data parallel approach. The alternative to this type of system may be called Local Data Scope (LDS). LDS has the quality that it potentially is much more communication efficient, but as a result is much harder to realize and not easily restructured for different algorithms. Another advantage to the GDS approach is that the beamform algorithm may be decomposed further, thus reducing Amdahl's fraction or the sequential bottleneck. In addition, GDS systems are likely to be much more fault tolerant since an agenda-parallel approach can decompose the beamform. One such agenda-parallel approach is steering decomposition where each node is in charge of beamsteering in one or a number of directions. These steering directions need not be static, however, and instead may be dynamically assigned throughout the nodes in the system. The results of this study are not algorithm dependent and may be generalized for conventional delay-and-sum frequency-domain beamforming or newer correlational techniques such as matched-field processing.

The sonar array is assumed to be a linearly distributed number of nodes each containing one or more transducers, a processing element, and a network connection, all tied together with a point-to-point inter-connection. Through broadcast mechanisms the network is able to provide 2

Mbps to each node's vector of data samples taken from 32 distributed nodes. The traffic calculations for non-broadcast networks and broadcast networks are shown in Equations E.1 and E.2, respectively.

$$(32 \text{ nodes}) \left(31 \frac{\text{destination packets}}{\text{node}} \right) \left(64 \frac{\text{bits}}{\text{packet}} \right) (1000 \text{ Hz}) \approx 64 \text{ Mbps}$$

Equation E.1

$$(32 \text{ nodes}) \left(1 \frac{\text{broadcast packets}}{\text{node}} \right) \left(64 \frac{\text{bits}}{\text{packet}} \right) (1000 \text{ Hz}) \approx 2 \text{ Mbps}$$

Equation E.2

The nodes fill up a large circular buffer in real time in parallel arrays. Thus, each node has a large matrix of sampled data as shown below in Figure E.2. A special array of pointers indicates the next storing location for each node. The figure shows time running down the page and samples are written asynchronously into each array. Some amount of synchronization is required to ensure proper beamform results, but this level is adjustable to practically any desired synchronization. Any beamform algorithm must be properly built to work within the bounds of the circular buffer. Currently, the fault-tolerant kernel does not inform the algorithm of the array bounds, but rather, the algorithm uses predetermined constants and operates within this domain.

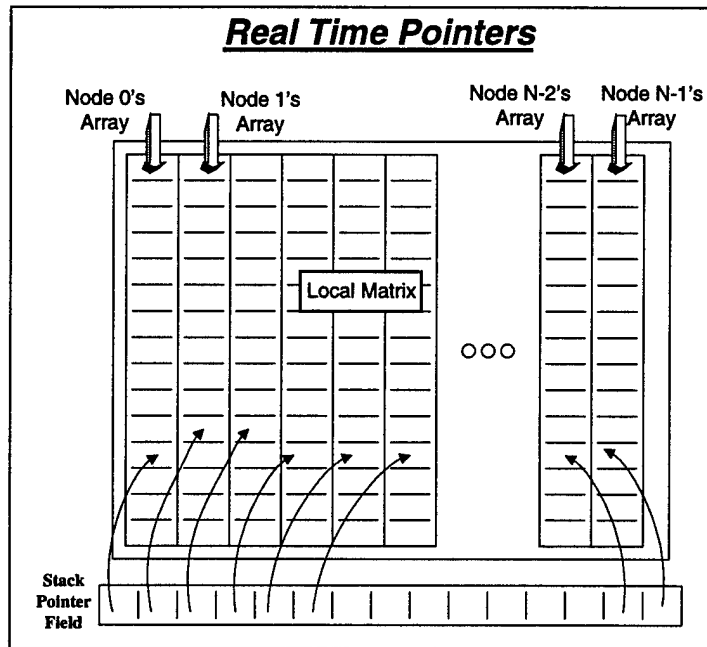


Figure E.2 - The Beamform Reduction Process. The beamform reduction operation occurs in two stages. The first stage reduces a large matrix into an array of summations. The second stage reduces this array into a sum of squares divided by the array size or power.

The beamform algorithm has the unique characteristic of being naturally fault tolerant. If data values or even nodes are lost, the algorithm may only be slightly affected, and valid results may still be gleaned from the system. The system can take advantage of this characteristic. Since the entire sample set may simply be approximately correct, the system may disregard the complications of ensuring data transmission. However, to minimize errors, these holes must be

patched slightly by padding them with zeroes. If not padded, the system is likely to use values from previous sample sets and loose synchronization, which may lead to corrupted results. This idea of dropping bits in real-time systems is by no means novel and solves a much more complicated problem with the greatest efficiency.

E.2. Data Padding Samples

Padding of the values is a simple task but requires some extra functionality in the network layers. The network must provide some sort of ordering mechanism, even if that is one bit. Of course, the more bits, the better the system will remain in synchronization and stay valid. The example software uses a 4-bit counter to order incoming data values; therefore, up to 15 data samples may be lost and correctly padded before losing synchronization. The software simply compares the counter value from the network packet with a local 4-bit accumulator. If they are in disagreement, then the distance between samples is determined and the proper number of samples is padded into the buffer. A different accumulator is local to each array of the data matrix; therefore, there are as many accumulators as there are real-time pointers or nodes. Figure E.3. shows the portion of code to perform this operation. The code in bold shows the particular instructions that perform the data padding capability. The *sample_number* array holds the current data number for each array's column including its own. The *PAD_NUMBER* might be an average or a true interpolation; however, for simplicity it was left as a zero constant.

```

if(clockcounter>(last_sample_time+(1/parameters.sample_frequency)))
{
    last_sample_time = clockcounter;
    Sample_Pack.sample_value = ad_convert(tempfile,0,&dummy);
    Sample_Pack.sample_number = sample_number[myrank];
    Sample_Pack.iteration = write_sample(Sample_Pack.sample_value,
        myrank,iteration,sample_matrix,sample_pointers);
    for(i = 0; i < number_of_nodes; i++)
    {
        if(i != myrank)
        {
            if(alive_nodes[i] == ALIVE)
                MPI_Send((void*)&Sample_Pack,sizeof(sample_package),
                    MPI_BYTE,i,SAMPLE_PAK,MPI_COMM_WORLD);
        }
        sample_number[myrank]++;
        sample_number[myrank] = sample_number[myrank] & SN_MASK;
    }
    for(i = 0; i < number_of_nodes; i++)
    {
        if(i != myrank)
        {
            if(alive_nodes[i] == ALIVE)
            {
                MPI_Iprobe(i,SAMPLE_PAK,MPI_COMM_WORLD,&flag,&status);
                if(flag)
                {
                    MPI_Recv((void*)&Sample_Pack,sizeof(sample_package),
                        MPI_BYTE,i,SAMPLE_PAK,MPI_COMM_WORLD,&status);
                    while(sample_number[i] != Sample_Pack.sample_number)
                    {
                        write_sample(PAD_NUMBER,i,iteration,
                            sample_matrix,sample_pointers);
                        sample_number[i] = sample_number[i] & SN_MASK;
                    }
                    write_sample(Sample_Pack.sample_value,i,iteration,
                        sample_matrix,sample_pointers);
                    sample_number[i]++;
                    sample_number[i] = sample_number[i] & SN_MASK;
                }
            }
        }
    }
}

```

Figure E.3 - Data Padding. Data padding is a method which will preserve synchronization of GDS systems without guaranteeing the delivery of data packets with a transport protocol.

Data padding may be graphically depicted as shown in Figure E.4. In this example, node 3 dropped a single data value, and the receiver pads it with a zero. Since the distance between the last sample number and the current number may be a number from 0 to *SN_MASK*, the protocol will detect multiple dropped packets up to *SN_MASK*.

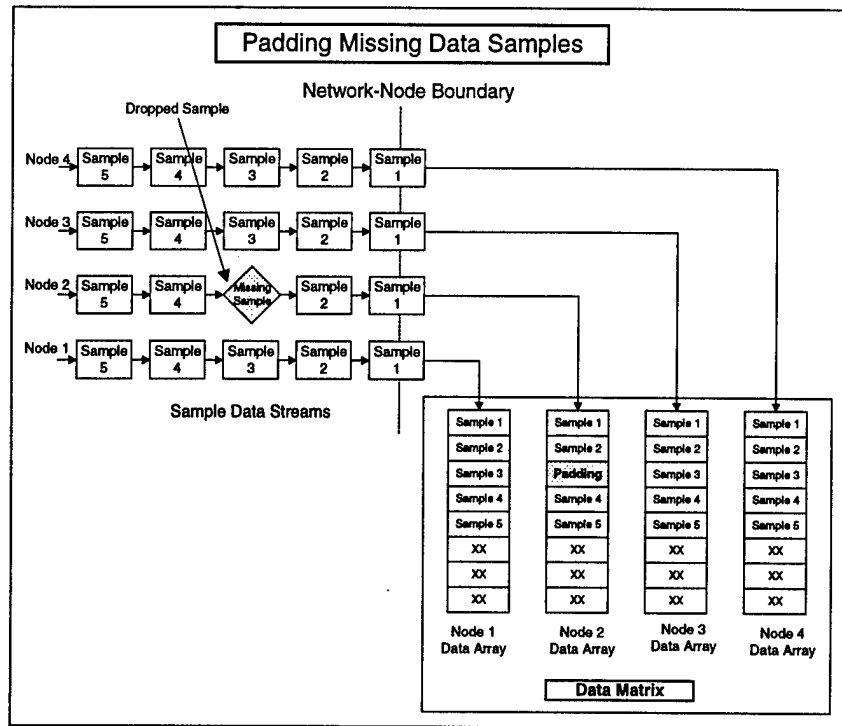


Figure E.4 - Padding Data Streams. The fault-tolerant kernel may pad lost data values with a zero or an average of the surrounding samples. Each node records a data packet number for each array of the circular matrix.

E.3. Automatic Repeat Requests

The second fault-tolerant measure aforementioned is more complicated. In some scenarios, it is imperative that a request (or network transaction) is successful. For instance, all slave nodes must correctly receive one command that resynchronizes the entire array. If one slave node was unsuccessful at receiving the transmission or responding to it, the resynchronization request must be broadcast again to all slave nodes. Only when acknowledge of the request from each of the nodes is returned to the master is the command considered successful. A simple stop-and-wait protocol was created for each request type. Each request that requires this service provides its own ARQ buffering mechanism for retransmission; therefore, a stop-and-wait system is not quite analogous. Figure E.5 shows the request response model for this approach. Beamform request, resynchronization request, and ping request each use this model to ensure stability.

E.4. Reinitialize Request

The reinitialize request may be called periodically to ensure proper synchronization of the global matrices. Reinitializing returns many variables, such as ARQ status variables and other timing variables, to their zero state and of course zeroes out the entire data matrix and sample pointers. One property of the reinitialize request, which is different from the other requests, is that if one or more nodes do not acknowledge the request, it must be resent to all of the nodes. If this measure is not taken and a reinitialize request is sent just to the nodes that did not respond, the synchronization of these nodes will lag all of the other nodes. So the reinitialize request is sent until every alive node returns an acknowledge.

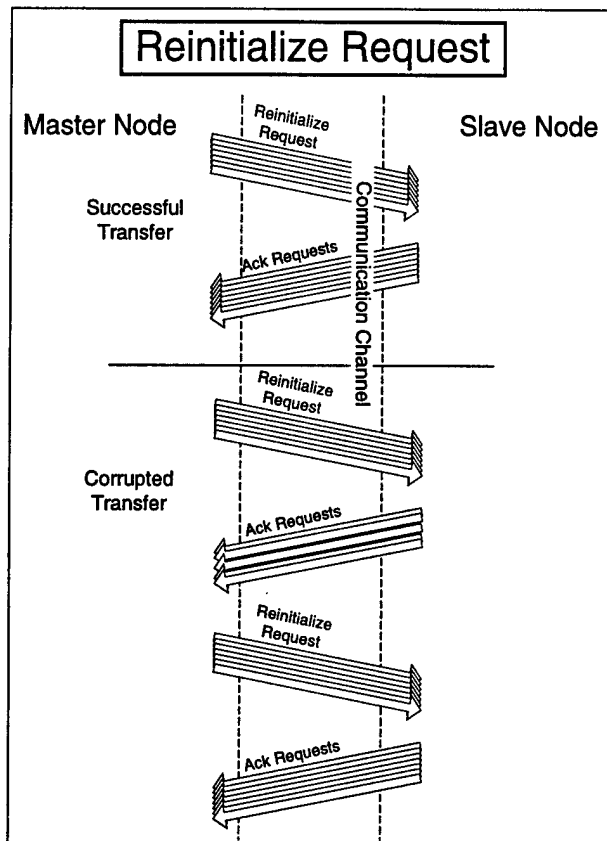


Figure E.5 - Reinitialize Request. One of the system messages which takes advantage of the ARQ is the reinitialize request. In its case, however, if one node fails to respond everyone must be reinitialized again.

E.5. Ping (Check Status) Request

The ping mechanism uses the request-acknowledge paradigm in another manner. In its case, the acknowledge indicates that the slave node is still operational. The master periodically makes ping requests to each node and tabulates the results. These results are then forwarded to each of the nodes so that they have an image of the current status. This table is also used when creating an agenda for the beamform algorithm. The ping request, if unacknowledged, makes repeated attempts to contact the slave node. If after a few requests the node is unresponsive, the slave will be cut off from the system. Each node keeps a record, which is continually updated by the master node, of which nodes are still alive. When each of the records are updated, the nodes will stop sending data to that node, the master will stop sending requests to that node, and any message generated by that node will be refused and sunk by the destined address. These measures will help to ensure that if the down node suddenly becomes active again, it will not try and send insane requests to other healthy nodes. Updating the nodes also performs the important function of zeroing out the data column of the defective node. This will prevent beamform solutions from using old data without having to do a complete reinitialize.

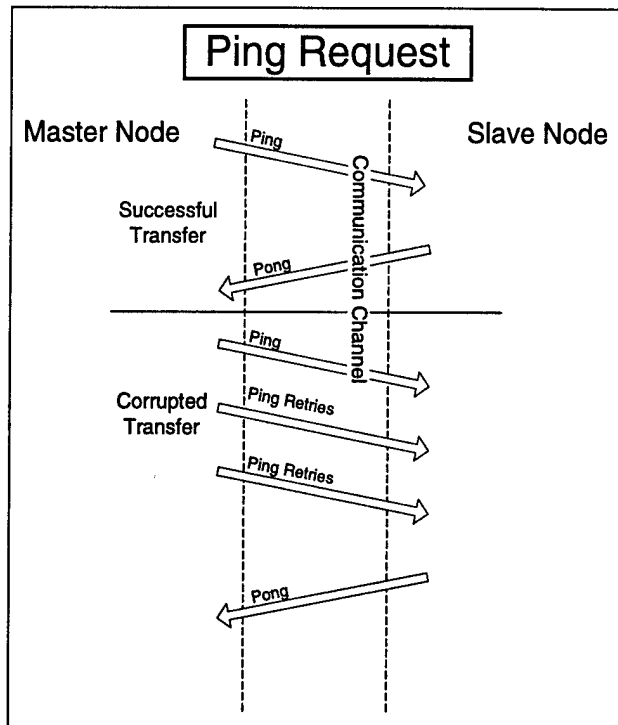


Figure E.6 - The Ping Request. The ping request checks the status of each node. If after a few attempts a node fails to respond to any of the ping requests, that node will be cutoff from communication to other node, and each of the nodes will be updated with an update system request.

E.6. Beamform Request

Beamform requests, like pings, may be selectively acknowledged. This implies that the beamform solutions do not necessarily have to occur at the same time or on the same set of data. Although the true solution does require this, if one beamform request operates on data that is one or two data samples behind, the solution is still approximately correct. In fact, with steering decomposition, the sample sets may even be orthogonal and still represent an accurate solution. This situation is not valid only if there is great flux in the incoming signals. The ARQ mechanism works exactly like the ping-pong mechanism and is coded similarly. Figure E.7 below shows an example beamform request transaction. Note that beamform results are not acknowledged, but they could be (via a simple extension to the slaves' node programming) if desired since they have the ARQ mechanism as well. However, it is not necessary that each beamform request result be successful since the master node may simply infer information from the results of its neighbors. In addition, in a real-time system, a missed piece of data has little effect.

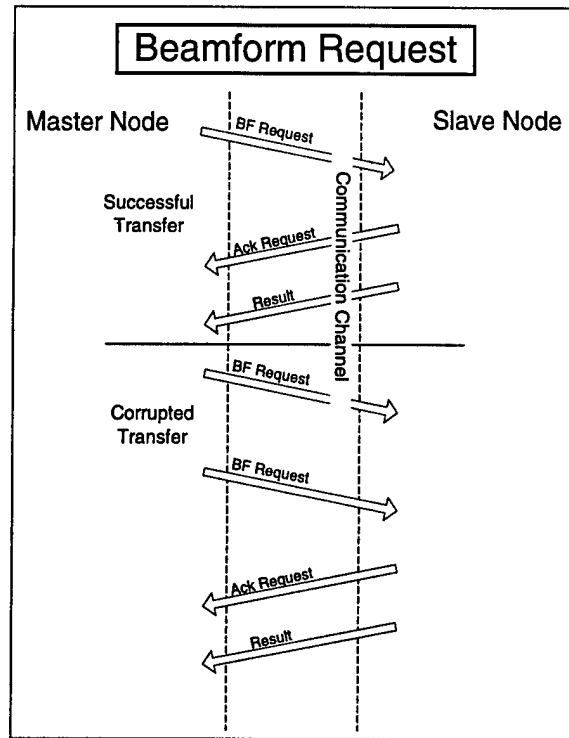


Figure E.7 - The Beamform Request. The beamform request occurs in two stages. The first stage immediately sends an acknowledge of the request. This ensures that a new beamform request may be created immediately if a node did not respond. The second stage is the result of the beamform request and may take an arbitrarily long period.

E.7. Arbitrate New Master

The last fault-tolerant measure built into the GDS kernel was a mechanism for timing the master node requests. If the master fails and stops sending requests then each node will start timing and arbitrate to become master after a specific period. The current arbitration is quite simple and works as follows. During initialization, the node with the lowest rank becomes the master. After each master node failure, the node with the lowest rank number becomes the new master. This process may continue all the way until there are only two nodes left, at which point any results taken from the array have little fidelity. This arbitration is likely to be much more complicated because the communication up-link is not likely to be present at every node and may instead be ported to a specific few. However, if the link is located on a boundary at a few of the nodes, this arbitration scheme would be sufficient. After the arbitration of master status is complete, the new master then sends a reinitialization request to each of the other nodes. A portion of code that shows the arbitration for becoming master is shown as Figure E.8 below. The bold text shows the actual instructions that provide this arbitration.

```

if(master_timeout > (2*setsize))
{
    new_master = 0;
    alive_nodes[last_master] = 0;
    master_timeout = 0;
    for(i = 0; i <= myrank; i++)
    {
        if(alive_nodes[new_master] == 0)
        {
            new_master++;
        }
    }
    if(new_master == myrank)
    {
        printf("Master Died - Node[%d] is the new Master\n",myrank);
        job_scheduler(setsize,number_of_nodes,myrank,filename,
            sample_matrix,sample_pointers,clock_resolution,
            parameters.sample_frequency,alive_nodes);
    }
}

```

Figure E.8 - Master Arbitration. If the master node goes down an arbitration scheme must resolve who the new master node will be. The new master is simply the alive node with the lowest rank.

E.8. Verification of GDS Fault-Tolerant Kernel

A complicated test was created to test the verification of the GDS kernel. This test injected faults into the system such as a node failure or master node failure. In addition, the reinitialize request was periodically called to test how well the array could remain in synchronization. Below is the text output of various phases of the system during these fault-injections. The output is an abridged version of the data file *fault_injection_test1.dat* which may be obtained by contacting the HCS Research Laboratory.

Abridged Output of Kernel Test with Fault-Injection

Clock (0.000104) integrally adjusted to samptime...

Request Reinitialize

Node[1] acknowledged reinitialize request
Node[2] acknowledged reinitialize request
Node[3] acknowledged reinitialize request
Node[4] acknowledged reinitialize request
Node[5] acknowledged reinitialize request
Node[6] acknowledged reinitialize request
Node[7] acknowledged reinitialize request

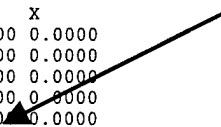
Ping All Nodes

Node[1] acknowledged PING
Node[2] acknowledged PING
Node[3] acknowledged PING
Node[4] acknowledged PING
Node[5] acknowledged PING
Node[6] acknowledged PING
Node[7] acknowledged PING

Request Beamform 1

```
0.0000 0.5878 0.9511 0.9511 0.5878 0.0000 0.5878 0.9511
0.3090 0.3090 0.8090 1.0000 0.8090 0.3090 0.3090 0.8090
0.5878 0.0000 0.5878 0.9511 0.9511 0.5878 0.0000 0.5878
1.0000 0.8090 0.3090 0.3090 0.8090 1.0000 0.8090 0.3090
0.9511 0.9511 0.5878 0.0000 0.5878 0.9511 0.9511 0.5878
0.8090 1.0000 0.8090 0.3090 0.3090 0.8090 1.0000 0.8090
0.5878 0.9511 0.9511 0.5878 0.0000 0.5878 0.9511 0.9511
0.3090 0.8090 1.0000 0.8090 0.3090 0.3090 0.8090 1.0000
X      X      X      X      X      X      X      X
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
```

Matrix Zeroed
at startup and
when reinitialized

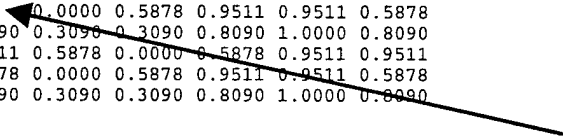


Node[1] acknowledged beamform request
Node[2] acknowledged beamform request
Node[3] acknowledged beamform request
Node[4] acknowledged beamform request
Node[5] acknowledged beamform request
Node[6] acknowledged beamform request
Node[7] acknowledged beamform request

Request Beamform 4

```
0.0000 0.5878 0.3090 0.9511 0.5878 0.0000 0.5878 0.9511
0.3090 0.3090 0.5878 1.0000 0.8090 0.3090 0.3090 0.8090
0.5878 0.0000 0.8090 0.9511 0.9511 0.5878 0.0000 0.5878
0.5878 0.0000 0.8090 0.9511 0.9511 0.5878 0.0000 0.5878
0.8090 0.3090 0.9511 0.8090 1.0000 0.8090 0.3090 0.3090
0.9511 0.5878 1.0000 0.5878 0.9511 0.9511 0.5878 0.0000
1.0000 0.8090 0.9511 0.3090 0.8090 1.0000 0.8090 0.3090
0.9511 0.9511 X 0.0000 0.5878 0.9511 0.9511 0.5878
0.8090 1.0000 0.8090 0.3090 0.3090 0.8090 1.0000 0.8090
0.5878 0.9511 0.9511 0.5878 0.0000 0.5878 0.9511 0.9511
0.9511 0.9511 0.5878 0.0000 0.5878 0.9511 0.9511 0.5878
0.8090 1.0000 0.8090 0.3090 0.3090 0.8090 1.0000 0.8090
```

Node 2
fails after 3rd
beamform request



Ping All Nodes

Node[1] acknowledged beamform request
Node[3] acknowledged beamform request
Node[4] acknowledged beamform request
Node[5] acknowledged beamform request
Node[6] acknowledged beamform request
Node[7] acknowledged beamform request
Node[1] acknowledged PING
Node[3] acknowledged PING
Node[4] acknowledged PING
Node[5] acknowledged PING
Node[6] acknowledged PING
Node[7] acknowledged PING
PingPong Failed - Node[0] checking #Pings = 0
PingPong Failed - Node[0] checking #Pings = 1
PingPong Failed - Node[0] checking #Pings = 2
PingPong Failed - Node[0] checking #Pings = 3
PingPong Failed - Node[0] checking #Pings = 4
Node 2 is dead

Master sends
Pings to Node 2
Discovers it dead

Update local node table

Master Died - Node[1] is the new Master
Request Beamform 1

Master Fails -
Nodes arbitrate
for master

X	0.3090	X	1.0000	0.8090	0.3090	0.3090	0.8090
0.0000	0.0000	0.0000	0.9511	0.9511	0.5878	0.0000	0.5878
0.0000	0.3090	0.0000	0.8090	1.0000	0.8090	0.3090	0.3090
0.0000	0.9511	0.0000	0.5878	0.0000	0.5878	0.9511	0.9511
0.0000	0.8090	0.0000	0.8090	0.3090	0.3090	0.8090	1.0000
0.0000	0.5878	0.0000	0.9511	0.5878	0.0000	0.5878	0.9511
0.0000	X	0.0000	X	X	X	X	X
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Node 0 (Old master)
zeroed

Request Beamform 5

X	0.9511	X	0.0000	0.5878	0.9511	0.9511	0.5878
0.0000	1.0000	0.0000	0.3090	0.3090	0.8090	1.0000	0.8090
0.0000	0.9511	0.0000	0.5878	0.0000	0.5878	0.9511	0.9511
0.0000	1.0000	0.0000	0.3090	0.3090	0.8090	1.0000	0.8090
0.0000	0.9511	0.0000	0.5878	0.0000	0.5878	0.9511	0.9511
0.0000	0.8090	0.0000	0.8090	0.3090	0.3090	0.8090	1.0000
0.0000	0.5878	0.0000	0.9511	0.5878	0.0000	0.5878	0.9511
0.0000	X	0.0000	X	X	X	X	X
0.0000	1.0000	0.0000	0.3090	0.3090	0.8090	1.0000	0.8090
0.0000	0.9511	0.0000	0.5878	0.0000	0.5878	0.9511	0.9511
0.0000	0.8090	0.0000	0.8090	0.3090	0.3090	0.8090	1.0000
0.0000	0.3090	0.0000	1.0000	0.8090	0.3090	0.3090	0.8090
0.0000	0.0000	0.0000	0.9511	0.9511	0.5878	0.0000	0.5878
0.0000	0.3090	0.0000	0.8090	1.0000	0.8090	0.3090	0.3090
0.0000	0.5878	0.0000	0.5878	0.9511	0.9511	0.5878	0.0000
0.0000	0.8090	0.0000	0.3090	0.8090	1.0000	0.8090	0.3090

Node[3] acknowledged beamform request
Node[4] acknowledged beamform request
Node[5] acknowledged beamform request
Node[6] acknowledged beamform request
Node[7] acknowledged beamform request
Request Beamform 6

Node 2
zeroed

```

Clock (0.000104) integrally adjusted to sampletime...

```

```
Update local node table
```

```
Master Died - Node[3] is the new Master
```

Request Reinitialize

```
Node[4] acknowledged reinitialize request
```

```
Node[4] acknowledged reinitialize request
Node[5] acknowledged reinitialize request
```

```
Node[5] acknowledged reinitialize request
Node[6] acknowledged reinitialize request
```

```
Node[6] acknowledged reinitialize request
Node[7] acknowledged reinitialize request
```

Ping All Nodes

```
Node[4] acknowledged PING
```

```
Node[4] acknowledged PING
Node[5] acknowledged PING
```

```
Node[5] acknowledged PING
Node[6] acknowledged PING
```

```
Node[7] acknowledged PING
```

Request Beamform 1

X	X	X	0.9511	0.9511	0.5878	0.0000	0.5878
0.0000	0.0000	0.0000	0.8090	1.0000	0.8090	0.3090	0.3090
0.0000	0.0000	0.0000	0.5878	0.9511	0.9511	0.5878	0.0000
0.0000	0.0000	0.0000	0.8090	0.3090	0.3090	0.8090	1.0000
0.0000	0.0000	0.0000	0.9511	0.5878	0.0000	0.5878	0.9511
0.0000	0.0000	0.0000	1.0000	0.8090	0.3090	0.3090	0.8090
0.0000	0.0000	0.0000	X	X	X	X	X
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Node[7]	acknowledged	beamform request					
Node[4]	acknowledged	beamform request					
Node[5]	acknowledged	beamform request					
Node[6]	acknowledged	beamform request					
Request	Beamform 2						

X	X	X	0.3090	0.3090	0.8090	1.0000	0.8090
0.0000	0.0000	0.0000	0.5878	0.0000	0.5878	0.9511	0.9511
0.0000	0.0000	0.0000	0.8090	0.3090	0.3090	0.8090	1.0000
0.0000	0.0000	0.0000	0.5878	0.0000	0.5878	0.9511	0.9511
0.0000	0.0000	0.0000	0.8090	0.3090	0.3090	0.8090	1.0000
0.0000	0.0000	0.0000	0.9511	0.5878	0.0000	0.5878	0.9511
0.0000	0.0000	0.0000	1.0000	0.8090	0.3090	0.3090	0.8090
0.0000	0.0000	0.0000	X	X	X	X	X
0.0000	0.0000	0.0000	0.5878	0.0000	0.5878	0.9511	0.9511
0.0000	0.0000	0.0000	0.8090	0.3090	0.3090	0.8090	1.0000
0.0000	0.0000	0.0000	0.9511	0.5878	0.0000	0.5878	0.9511
0.0000	0.0000	0.0000	0.8090	1.0000	0.8090	0.3090	0.3090
0.0000	0.0000	0.0000	0.5878	0.9511	0.9511	0.5878	0.0000
0.0000	0.0000	0.0000	0.3090	0.8090	1.0000	0.8090	0.3090
0.0000	0.0000	0.0000	0.0000	0.5878	0.9511	0.9511	0.5878

Performance tests were taken against a baseline GDS system, which may show a rough estimation of the performance slowdown due to the extra fault-tolerant features. Note that during this test, fault-injections were turned off. The results, then, are representative of a healthy array and not of transient periods. However, all of the kernel recovery services are, of course, still present, thus adding overhead to the system. The system exhibits high efficiency implying many services may be provided if their frequency of interruption with the system is sparse. In this particular test, all of the fault-injection parameters were turned off, but fault-tolerant services such as frequent ping requests were still performed.

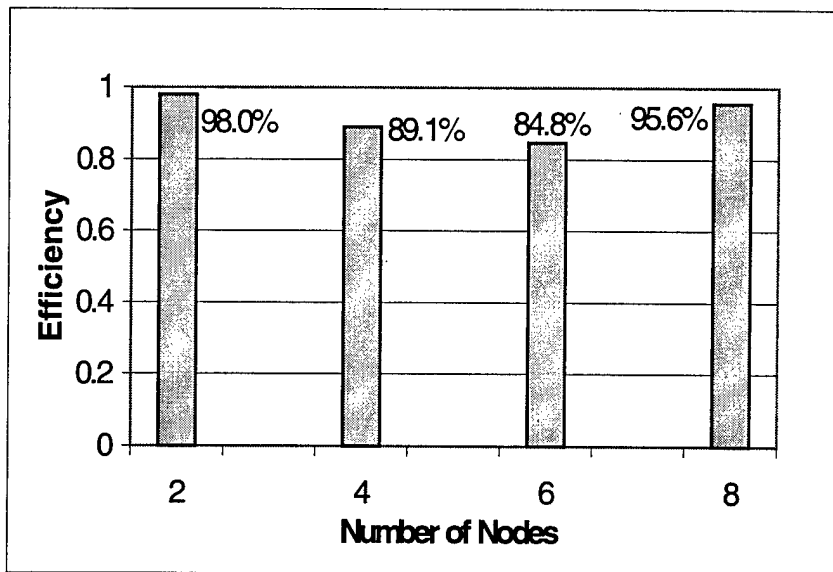


Figure E.9 - Efficiency Test. The efficiency of the fault-tolerant kernel surprisingly was very high. This indicates that these services may be provided at low cost to the performance on the network.

E.9. Conclusions

The software for the GDS kernel was built so that any beamform algorithm, conventional or otherwise, may be simply written into the simulator. GDS is a communication framework that distributes data to each node so that data is logically a shared file. However, the system may be optimized for its domain. For instance, if operating in the frequency domain, intuitively we see that the data should be transformed before broadcasting to each node. This optimization is used because parallelism of the data is inherent to the system and missing this step makes inefficient use of the hardware. In general, any preprocessing step such as filtering or transforming should be implemented before the broadcast distribution by the GDS nodes.

The results of the fault-tolerant kernel suggest that many mechanisms may be used in order to maintain a stable system at reasonable cost to the entire system's processing capability and to the communication channel. Tests showed that the five fault-tolerant mechanisms only lowered system performance by 10%. This factor is reasonable, and the final prototype may be over designed to compensate for this loss.

REPORT DOCUMENTATION PAGEForm Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave
Blank)2. REPORT DATE
3 Feb 983. REPORT TYPE AND DATES COVERED
Annual (1 Jan 97 - 31 Dec 97)

4. TITLE AND SUBTITLE

Parallel and Distributed Computing Architectures and Algorithms for Fault-Tolerant Sonar Arrays
(Annual Report #2)

5. FUNDING NUMBERS

N00014-97-1-0229

6. AUTHORS

A. George, R. Fogarty, J. Garcia, K. Kim, J. Markwell, M. Miars, and S. Walker

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

HCS Research Laboratory
Dept. of Electrical and Computer Engineering, University of Florida
PO Box 116200, 320 Larsen Hall
Gainesville, FL 32611-62008. PERFORMING ORGANIZATION REPORT
NUMBER

HCS-TR-98-1

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Ballston Centre Tower One
800 N Quincy Street
Arlington, VA 22217-566010. SPONSORING / MONITORING AGENCY
REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report summarizes the progress and results of the second of a three-year study whose goal is the use of fault-tolerant distributed and parallel processing techniques to decrease the cost and improve the performance and reliability of large, disposable sonar arrays. In this second phase, tasks have concentrated on the design, development, analysis and evaluation of conventional and basic split-aperture beamforming algorithms. A broad assortment of parallel algorithms and programs for FFT beamformers have been completed, and their performance evaluated via a cluster testbed and via rapid virtual prototyping capabilities derived from new network architecture models. These models include a wide variety of network protocols centered around unidirectional, ring, and bidirectional topologies. In addition, new emphasis has begun with split-aperture conventional beamforming and initial results indicate a significant potential for performance improvement through parallel processing. Finally, the architecture for the hardware prototype has been developed, and work has begun on its construction and that of its software system.

14. SUBJECT TERMS

distributed computing; parallel computing; computer networks; sonar arrays; beamforming algorithms;
fault-tolerant computing

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION
OF REPORT
UNCLASSIFIED18. SECURITY CLASSIFICATION
OF THIS PAGE
UNCLASSIFIED19. SECURITY CLASSIFICATION
OF ABSTRACT
UNCLASSIFIED20. LIMITATION OF
ABSTRACT